

ARBEITSBERICHTE

DES INSTITUTS FÜR MATHEMATISCHE
MASCHINEN UND DATENVERARBEITUNG

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER : H. Billing, W. Händler, U. Herzog,

F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,

H. Niemann, D. Seitzer, H. J. Schnelder

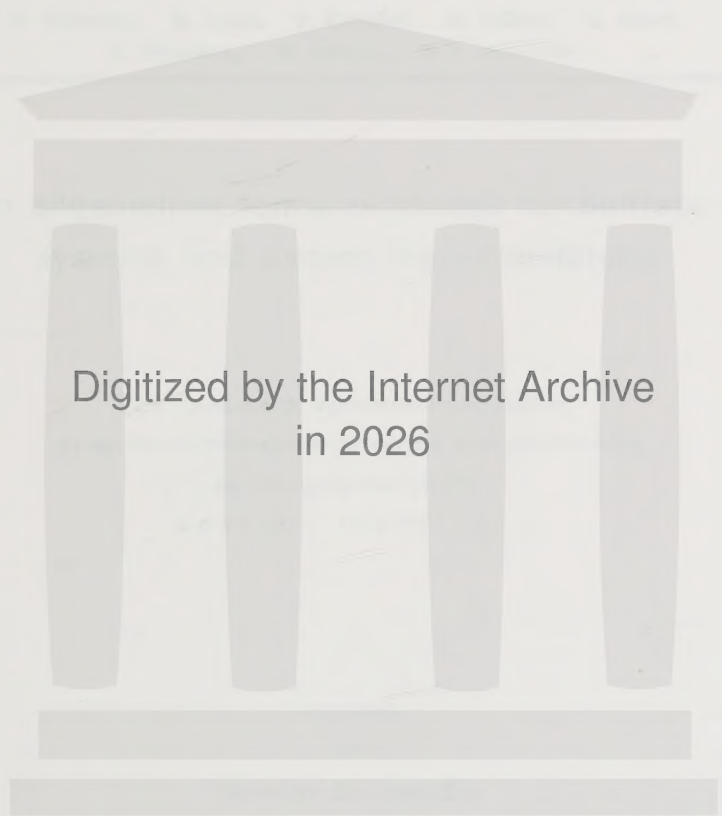
Ein allgemeines formales Modell für Betriebs-
systeme und dessen Implementierung

Sawwas Keramidis

Dissertation

BAND 10 · NUMMER 13 · ERLANGEN · SEPTEMBER 1977

1977 № 358



Digitized by the Internet Archive
in 2026

ARBEITSBERICHTE

DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER : H. Billing, W. Händler, U. Herzog,

F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,

H. Niemann, D. Seitzer, H.J. Schneider

Ein allgemeines formales Modell für Betriebs- systeme und dessen Implementierung

Dem Fachbereich Ingenieurwissenschaften
an der Friedrich-Alexander-Universität Erlangen-Nürnberg
zur Erlangung des Grades
D o k t o r - I n g e n i e u r

vorgelegt

von

Sawwas Keramidis

Dissertation

BAND 10 · NUMMER 13 · ERLANGEN · SEPTEMBER 1977

ARBEITSGERICHT
DES INSTITUTS FÜR MATHEMATISCHE
MASCHINEN UND DATENVERARBEITUNG
WIRTSCHAFTS UNIVERSITÄT ERLANGEN-NÜRNBERG
LEHRSTUHL FÜR MASCHINEN- UND DATENVERARBEITUNG
LEHRSTUHL FÜR MASCHINEN- UND DATENVERARBEITUNG
LEHRSTUHL FÜR MASCHINEN- UND DATENVERARBEITUNG

Ein abstraktes formales Modell für Betriebs-
systeme und dessen Implementierung

von Dr. rer. oec. habil. Dr. rer. oec. habil.
Dr. rer. oec. habil. Dr. rer. oec. habil.
Dr. rer. oec. habil. Dr. rer. oec. habil.
Dr. rer. oec. habil. Dr. rer. oec. habil.
Dr. rer. oec. habil. Dr. rer. oec. habil.



Als Dissertation genehmigt vom
Fachbereich Ingenieurwissenschaften
an der Universität Erlangen-Nürnberg

Tag der Einreichung: 19.4.1977

Tag der Promotion: 28.6.1977

Dekan: Prof. Dr. H.J. Schneider

Berichterstatter: Prof. Dr. F. Hofmann, Prof. Dr. H.J. Schneider

Herr Professor Dr. F. Hofmann hat diese Arbeit angeregt und sie während ihrer gesamten Entwicklung mit großem Interesse verfolgt. Dafür und für seine anregenden Diskussionen und konstruktive Kritik möchte ich ihm an dieser Stelle herzlich danken.

Auch Frl. E. Roth, die die Reinschrift dieser Arbeit erstellte, möchte ich hier meinen Dank aussprechen.

<u>Inhalt</u>	<u>Seite</u>
Zusammenfassung	1
0. Einführung	2
0.1 Allgemeine Bemerkungen	2
0.2 Ziele und Zwecke dieser Arbeit	5
0.3 Bisherige Arbeit über dieses Gebiet	8
1. Prozesse und Prozeßsysteme	11
1.1 Sequentielle Prozesse	11
1.2 Systeme von Prozessen	19
2. Eine universale Basis für Betriebssysteme	47
2.1 Einleitung	47
2.2 Konstruktion einer Betriebsmittel-Maschine	49
3. Beispiele und Anwendungen	82
3.1 Modell eines Auskunftssystems	82
3.2 Das zwei Erzeuger-, zwei Verbraucher-Problem	91
3.3 Das Alarmclock-Problem	94
3.4 Nichtverträglichkeitsrelationen	103
3.5 Die Behandlung von Hardware-Einheiten mit einer Betriebsmittel-Maschine	105
4. Ein Konzept zur Implementierung der Synchronisation von Prozessen in einer Betriebsmittel-Maschine	114
Literatur	134

Zusammenfassung:

In dieser Arbeit wird ein allgemeines formales Modell entwickelt, welches die wesentlichen Eigenschaften von Betriebssystemen (z.B. "Starten", "Blockieren" und "Verklemmen" von Prozessen) unmittelbar zum Ausdruck bringen läßt. Auch ein Konzept zur effektiven Implementierung dieses Modells wird hier vorgestellt.

Es stellt sich heraus, daß das Problem der Existenz von Verklemmungen in einem beliebig vorgegebenen Betriebssystem nur partiell lösbar ist. Das Problem der Nichtexistenz von Verklemmungen in einem beliebig vorgegebenen Betriebssystem ist nicht einmal partiell lösbar. Weiterhin wird eine eingeschränkte Klasse von Betriebssystemen angegeben, für die beide oben genannten Probleme lösbar sind.

Für Betriebssysteme typische Probleme werden mit Hilfe des hier entwickelten Modells dargestellt.

2. Einführung

2.1 Allgemeine Bemerkungen

Die von einem Betriebssystem abverlangten Leistungen haben erheblich zugenommen, seitdem Mehrprogramm- und Mehrprozessorbetriebe realisiert wurden.

Der Zugriff auf periphere Geräte mit mechanisch bewegten Teilen ist um einige Zehnerpotenzen langsamer als der Zugriff auf den Arbeitsspeicher eines Rechners. Wartet nun ein auf einem Ein-Prozessor-Rechner laufendes Programm auf Fertigmeldung eines Gerätes, so ist es sinnvoll, während dieser Zeit den Prozessor einem anderen Programm zuzuordnen (Mehrprogramm-Betrieb). Das Betriebssystem muß jetzt unter anderem die Prozessor-Umschaltung vornehmen und dafür sorgen, daß keine Konfliktsituationen entstehen, die dann auftreten können, wenn verschiedene Programme gleichzeitig auf seriell benutzbare Einheiten (z.B. Geräte, Unterprogramme, Warteschlangen, Nachrichten) zugreifen. Bereits beim Mehrprogramm-Betrieb scheint es zweckmäßig zu sein, die Erledigung mancher Aufträge in mehrere selbständig formulierbare und zeitlich parallel ausführbare Aktivitäten (Prozesse genannt) zu zerlegen.

Informell kann man einen Prozeß als eine zusammengehörige Sequenz von Instruktionen auffassen, die ausgeführt werden kann, falls ein Prozessor zur Verfügung steht. Hat man mehrere Prozessoren zur Verfügung (Mehrprozessorbetrieb), so kann jedem Prozessor ein Prozeß zugeordnet werden. Macht man keine Einschränkungen bzgl. der relativen Geschwindigkeiten mit denen die einzelnen Prozesse ihre Operationen ausführen, so

spricht man auch von asynchron ablaufenden Prozessen. Ein Prozeß kann sowohl private als auch gemeinsame Größen manipulieren, wobei oft das Fortschreiten eines Prozesses vom Zustand gemeinsamer Größen abhängig ist. Diese Größen werden hier Betriebsmittel genannt und symbolisieren z.B. Geräte, Nachrichten, Warteschlangen usw. Die Situation, bei der mehrere Prozesse durch asynchrone Arbeitsweise an der Lösung eines Problems beteiligt sind, heißt Parallelität. Liegt zu einem gegebenen Zeitpunkt t ein Systemzustand vor, von dem aus ein Prozeß P nicht fortfahren kann, so heißt P blockiert und der entsprechende Zustand Blockierungszustand von P . Zu einem späteren Zeitpunkt kann sich aber der Systemzustand so geändert haben, daß P doch fortschreiten kann. Ist dies nicht möglich, so heißt der Zustand auch Verklemmungszustand für P und P heißt in diesem Zustand verklemmt. Eine typische Eigenschaft von Prozessen in der Betriebsprogrammierung ist, daß die einzelnen Prozesse zyklisch arbeiten. Bei korrekter Arbeitsweise des Systems darf kein Prozeß in einen Verklemmungszustand geraten. In der Praxis gibt es verschiedene Einstellungen zu dem Verklemmungsproblem. Man kann den Standpunkt vertreten, daß ein Betriebssystem so konstruiert werden muß, daß keine Verklemmungen möglich sind. Dies ist aber in der Regel mit einer verminderten Auslastung der Hardware-Betriebsmittel verbunden. In Betriebssystemen, die Verklemmungsmöglichkeiten zulassen, gibt es wiederum verschiedene Strategien zum Umgang mit Verklemmungen: 1) Man sorgt dafür, daß keine Verklemmung eintritt (Vermeidung von Verklemmungen), oder 2) man erlaubt zwar Verklemmungen, sorgt aber dafür, daß diese Situationen rechtzeitig erkannt werden (Erkennung von Verklemmungen). Maßgebend für die verschiedenen Standpunkte sind Fragen, die den mit der jeweiligen Vorgehensweise verbundenen organisatorischen Aufwand anbetreffen.

Aus Effizienzgründen entzieht man einem blockierten Prozeß den Prozessor. Kann ein blockierter Prozeß wieder fortfahren, so ordnet man diesem einen Prozessor zu. In diesem Fall spricht man vom "Wecken" eines (blockierten) Prozesses. Dieses "Wecken" von Prozessen ist nicht problemlos. Sei z.B. zu einem Zeitpunkt t die Umgebung eines Systems so verändert, daß nun jeder der bis zu diesem Zeitpunkt blockierten Prozesse aus $P_1, P_2, \dots, P_n$ fortfahren (gestartet werden) kann. Nach Ausführung eines dieser Prozesse sind aber unter Umständen die Startbedingungen mancher dieser Prozesse nicht mehr erfüllt. Das "Blockieren" bzw. "Wecken" von asynchronen Prozessen bildet den Kern der Synchronisation in einem Betriebssystem. Bezüglich des "Weckens" von Prozessen kann man wiederum verschiedene Standpunkte vertreten:

- 1) Ändert ein Prozeß P seine Umgebung so, daß die blockierten Prozesse $P_1, P_2, \dots, P_n$ gestartet werden können, so entscheidet P , welche Prozesse auch wirklich gestartet werden.
- 2) Der Prozeß P ändert seine Umgebung ohne sich um blockierte Prozesse zu kümmern. Die Synchronisationsarbeit wird von einer dem System inhärenten Instanz (Monitor oder Überwacher genannt) übernommen.

Bei der zweiten Methode sind mehr Umschaltungsverfahren erforderlich. Dafür tritt bei der ersten Vorgehensweise folgende Schwierigkeit auf: Prozeß P_1 kann nur dann Prozeß P_2 unmittelbar wecken, wenn P_1 die Startbedingungen von P_2 kennt. Eine Änderung der Startbedingungen von P_2 , aus welchen Gründen auch immer es geschieht, zwingt zu einer Änderung von P_1 .

Die Synchronisation von asynchron ablaufenden Prozessen ist ein wichtiger Teil in der Theorie und Praxis der Betriebssystem-Konstruktion. Die Implementierung der Synchronisation muß vor allem effektiv sein, damit keine unnötig langen Wartezeiten für die blockierten Prozesse entstehen.

0.2 Ziele und Zwecke dieser Arbeit

Die wesentlichen Merkmale von Betriebssystemen sind das Starten und Blockieren von Prozessen, die Synchronisation von asynchron ablaufenden Prozessen sowie deren Verklemmung. Das Grobziel dieser Arbeit war die Angabe eines allgemeinen Modells, welches die oben genannten Merkmale unmittelbar zum Ausdruck bringen läßt (Adäquatheit). Die formale Erfassung dieser Merkmale bildet das eigentliche Ziel der Arbeit. Die Ermittlung eines Konzepts zur effektiven Implementierung der Synchronisation und die Beantwortung der für die Praxis wichtigen Fragen nach der Existenz bzw. Nichtexistenz von Verklemmungsmöglichkeiten in einem beliebig vorgegebenen Betriebssystem sind die Hauptzwecke dieser Arbeit.

Ausgangspunkte waren ein Vorentwurf von Prof.F.Hofmann für die formale Erfassung von Prozessen und Prozeßsystemen sowie Vorarbeiten über effektiv implementierbare Synchronisationsoperationen.

Zur Konstruktion eines allgemeinen Modells für Betriebssysteme wurden zuerst existierende Betriebssysteme untersucht. In jedem Betriebssystem werden im wesentlichen dieselben Probleme behandelt, selbst wenn dies mit verschiedenen Hilfsmitteln geschieht. Das Erkennen der Gemeinsamkeiten der verschiedenen Darstellungsformen und die Untersuchung der Beziehungen zwischen den ver-

schiedenen Komponenten von Betriebssystemen haben zur ersten formalen Erfassung der wesentlichen Bestandteile eines Betriebssystems als Prozeßsystem geführt (Kapitel 1).

Danach stellt sich die Frage, ob aus diesem Modell ein einfacheres abgeleitet werden kann, welches aber trotzdem universell und der Darstellung von Betriebssystemen adäquat ist. Dies wird im Kapitel 2 positiv beantwortet. Im selben Kapitel wird eine abstrakte Maschine mit den gewünschten Eigenschaften konstruiert, wobei andere Aspekte der Betriebsprogrammierung wie z.B. Datensicherung und Warteschlangenprobleme nicht explizit behandelt werden. Deshalb wird von einer Betriebsmittel- und nicht von einer Betriebssystem-Maschine gesprochen. Auch die oben genannten Aspekte lassen sich jedoch in dieses Modell einbauen. Die gleichzeitig durchgeführten Arbeiten über die effektive Implementierung der Synchronisation haben großen Einfluß auf die endgültige Form der Betriebsmittel-Maschine ausgeübt. Erst durch die Konstruktion dieser Maschine ist es möglich, Aussagen über das Problem der Existenz bzw. Nichtexistenz von Verklemmungen in beliebig vorgegebenen Betriebssystemen zu machen. Es zeigt sich, daß das Problem der Existenz von Verklemmung nicht lösbar ist, wohl aber partiell lösbar. Das Problem der Nichtexistenz von Verklemmungen ist nicht einmal partiell lösbar. Durch Einführung von Einschränkungen in der allgemeinen Form der Betriebsmittel-Maschine kann man diese Probleme lösbar machen. Eine solche Einschränkung wird im 3. Kapitel gemacht. Die dabei entstehenden Systeme könnte man auch als "Habermann'sche"-Systeme bezeichnen. Die bekanntesten Arbeiten von Habermann beziehen sich nämlich auf solche Systeme. In diesem Zusammenhang wer-

den neue Überlegungen aufgestellt und Aussagen für die oben genannten Systeme bewiesen. Weiterhin werden in diesem Kapitel einige für Betriebssysteme typische Probleme behandelt, darunter das 2-Erzeuger/2-Verbraucher-Problem. Es ist bereits bekannt, daß das letztere Problem weder mit den Vektor-Substitutions-Systemen noch mit verallgemeinerten Petri-Netzen gelöst werden kann ([55]). Je nach Aufgabenprofil kann man außerdem neue Größen (z.B. Variablen oder Relationen) einführen, so daß sich manche Darstellungen vereinfachen und die Implementierung effektiver wird. Diesbezügliche Beispiele werden im 3. Kapitel angegeben.

Im 4. Kapitel schließlich wird ein Konzept zur effektiven Implementierung des hier entwickelten Modells angegeben. Den zeitaufwendigsten Teil bei der Synchronisation bildet derjenige Teil des Betriebssystems, der folgendes Problem löst: Seien zu einem Zeitpunkt t manche gemeinsame Variablen verändert. Diese Änderung betreffe die bis zu diesem Zeitpunkt blockierten Prozesse $P_1, P_2, \dots, P_n$ in dem Sinne, daß die Startbedingung jedes dieser Prozesse von mindestens einer dieser veränderten Variablen abhängt. Es sollen die Prozesse aus $\{P_1, P_2, \dots, P_n\}$ herausgesucht werden, die auch gestartet werden können. Es wurde bereits erwähnt, daß zwei Prozesse P_1 und P_2 nicht immer dann gestartet werden können, wenn ihre Startbedingungen erfüllt sind. Durch das Starten von P_1 kann nämlich die Startbedingung von P_2 verletzt werden. Im vorliegenden Modell werden die Prozesse $P_1, P_2, \dots, P_n$ bereits bei ihrer Blockierung in eine lineare Liste eingetragen und zwar so, daß vor-

handene Prioritätsverhältnisse unter diesen Prozessen nicht verletzt werden (prioritätskonforme Anordnung). Findet eine Änderung gemeinsamer Variablen statt, so wird nur dann etwas unternommen, wenn dies einen Sinn hat. So hat es z.B. keinen Sinn, nach dem Belegen eines Gerätes den Versuch zu unternehmen, blockierte Prozesse zu wecken. Findet aber eine erfolgsversprechende Veränderung statt, so wird die lineare Liste $P_1, P_2, \dots, P_n$ nur einmal in dieser Reihenfolge untersucht. Die dabei ermittelten Prozesse können dann wirklich ohne gegenseitige Störung gestartet werden.

Die aus der Literatur bekannten Konzepte der Monitore bzw. der bedingten kritischen Abschnitte (siehe 0.3) gestatten nur eine weniger effektive Implementierung.

0.3 Bisherige Arbeiten über dieses Gebiet

Die bisherigen Arbeiten über parallel bzw. asynchron arbeitende Systeme kann man im wesentlichen in zwei Klassen einteilen:

1) die theoretisch orientierten und 2) die praxisorientierten Arbeiten. Die ersten (z.B. [45,46,48,49]) werden von anderen Intentionen geprägt als die vorliegende Arbeit. Sie behandeln in erster Linie nichtinterpretierte Modelle (Schemata) und konzentrieren sich auf Fragen der Determiniertheit eines Schemas bzw. der Äquivalenz von Schemata. Sie sind leider für Betriebssysteme von geringerer Bedeutung, da sie dafür keine adäquaten Modelle darstellen. Auch die Mc-Culloch-Pitts-Zellen ([57]) und die daraus ableitbaren Petri-Netze sowie die Vektor-Substitutions-Systeme können nur sehr umständlich zur Darstellung von Betriebssystemen eingesetzt werden.

Zu den praxisorientierten Arbeiten gehören die von Dijkstra mit den auf Semaphore definierten P- und V-Operationen. Für diese gilt aber dasselbe, was über Petri-Netze gesagt wurde. Die bedingten kritischen Abschnitte ([29]) und das Monitor-Konzept ([32]) sind heute die einzigen für die Praxis der Betriebssysteme interessanten Modelle. Sie gestatten auch für komplizierte Fälle eine relativ übersichtliche Darstellung und sind in dem Sinne vollständig, daß sich jedes lös-bare Synchronisations-Problem mit Hilfe dieser Modelle darstellen läßt ([1]). Ein Nachteil dieser Modelle ist die Tatsache, daß sie einer abstrakten Behandlung schlecht zugänglich sind. Ihre wesentliche Schwäche liegt vor allem darin, daß sie eine zeitaufwendige Implementierung erfordern. Bei der Implementierung des Konzeptes der bedingten kritischen Abschnitte wird z.B. das in 0.2 erwähnte Problem folgendermaßen behandelt: Die Prozesse $P_1, P_2, \dots, P_n$ werden in dieser Reihenfolge so lange hintereinander untersucht, bis ein Prozeß gefunden wird, dessen Startbedingung erfüllt ist. Man startet diesen Prozeß und wartet auf seine Fertigmeldung; anschließend entfernt man diesen Prozeß aus der Warteschlange. Mit der neuen Liste verfährt man ebenso, bis kein Prozeß mehr gestartet werden kann. In letzter Zeit wurde zwar ein Verfahren zur effizienteren Implementierung des Konzeptes der bedingten kritischen Abschnitte gefunden ([79]), trotzdem bleibt sie immer noch recht zeitaufwendig.

Bei der Implementierung des Monitor-Konzeptes treten zusätzlich implementierungsspezifische Einschränkungen auf, die mit dem eigentlichen Synchronisationsproblem nichts zu tun haben. Beendet z.B. ein Prozeß eine Operation und kann dadurch ein bisher blockierter Prozeß geweckt werden,

so muß der weckende Prozeß auf eine Meldung des geweckten Prozesses warten.

Das in dieser Arbeit entwickelte Modell dürfte somit insbesondere für Echt-Zeit-Betriebssysteme von Bedeutung sein.

1. Prozesse und Prozeßsysteme

1.1 Sequentielle Prozesse

1.1.1 Definition

Unter einem Prozeß wird das Quintupel $P=(A,D,R,a_0,D_0)$ verstanden, wobei für die verschiedenen Größen gilt:

A ist eine endliche, nichtleere Menge (Adreßmenge genannt),

D ist eine nichtleere Menge (Datenmenge genannt),

R ist eine Relation, $R \subseteq Q \times Q$, wobei $Q = A \times D$ ist.

R wird Prozeßrelation, Q Zustandsmenge genannt.

a_0 ist die Anfangsadresse ($a_0 \in A$) des Prozesses.

Für D_0 gilt $D_0 \subseteq D$. Die Menge der Anfangszustände ist $\{a_0\} \times D_0$.

1.1.2 Definition

Sei $q \in Q$ mit $q=(a,d)$. Mit $\pi_\alpha(q)$ soll der Adreßteil von q bezeichnet werden, d.h. $\pi_\alpha(q)=a$. Ist $\rho \in R$ mit $\rho=(q,q')$, dann wird $\pi_\alpha(\rho)$ definiert durch

$$\pi_\alpha(\rho) = (\pi_\alpha(q), \pi_\alpha(q')).$$

1.1.3 Definition

Die Relation $\sim_\alpha \subseteq R \times R$ wird folgendermaßen eingeführt:

$$\sim_\alpha = \{(\rho_1, \rho_2) / (\rho_1, \rho_2) \in R \times R \wedge \pi_\alpha(\rho_1) = \pi_\alpha(\rho_2)\}$$

Dies bedeutet, daß für beliebige $\rho_1 \in R$ und $\rho_2 \in R$ mit

$$\rho_1=(q_1,q'_1) \text{ und } \rho_2=(q_2,q'_2) \text{ die Beziehung } \rho_1 \sim_\alpha \rho_2$$

genau dann gilt, wenn

$$\pi_\alpha(q_1) = \pi_\alpha(q_2) \text{ und } \pi_\alpha(q'_1) = \pi_\alpha(q'_2) \text{ gilt.}$$

1.1.4 Satz

Die oben angeführte Relation $\sim_\alpha$ ist eine Äquivalenzrelation.

Beweis

Der Beweis ist trivial, da die Symmetrie, Reflexivität und Transitivität von $\sim_\alpha$ ohne weiteres aus Definition 1.1.3 folgen.

Mit $\rho|_{\sim_\alpha}$ wird die Äquivalenzklasse von $(\rho \in R)$ bezüglich $\sim_\alpha$ bezeichnet, d.h. $\rho|_{\sim_\alpha} = \{\rho' \mid \rho' \in R \wedge \rho \sim_\alpha \rho'\}$

Mit $R|_{\sim_\alpha}$ wird die Restklassenmenge von R bezüglich $\sim_\alpha$ bezeichnet; d.h. $R|_{\sim_\alpha} = \{\rho|_{\sim_\alpha} \mid \rho \in R\}$

Der Einfachheit wegen wird $\tilde{\rho}$ statt $\rho|_{\sim_\alpha}$ und $\tilde{R}$ statt $R|_{\sim_\alpha}$ geschrieben.

1.1.5 Satz

Die oben eingeführte Restklassenmenge $\tilde{R}$ ist endlich.

Beweis

Für zwei beliebige $\tilde{\rho}_i \in \tilde{R}$ und $\tilde{\rho}_j \in \tilde{R}$ gilt:

$$\forall \rho (\rho \in \tilde{\rho}_i \Rightarrow \pi_\alpha(\rho) = \text{constant} = (a_i, a_i')) ,$$

$$\forall \rho (\rho \in \tilde{\rho}_j \Rightarrow \pi_\alpha(\rho) = \text{constant} = (a_j, a_j')) .$$

Aus Definition 1.1.3 und Satz 1.1.4 folgt:

$$\tilde{\rho}_i \neq \tilde{\rho}_j \implies (a_i, a'_i) \neq (a_j, a'_j)$$

Da die Adressenmenge A endlich ist, können endlich viele Paare $(a, a') \in A \times A$ gebildet werden. Die Restklassenmenge $\tilde{R}$ ist also endlich.

Aufgrund dieses Beweises erscheint die Einführung einer neuen Relation sinnvoll.

1.1.6 Definition

Für je zwei $a_i \in A$ und $a_j \in A$ wird die Relation $O_{ij} \subseteq D \times D$ gemäß folgender Beziehung eingeführt:

$$O_{ij} = \left\{ (d, d') \mid (d, d') \in D \times D \wedge \exists \rho \in R (\rho = ((a_i, d), (a_j, d'))) \right\}$$

1.1.7 Satz

Die Menge aller nichtleeren O_{ij} -Relationen sei mit O bezeichnet. Dann sind die Mengen O und $\tilde{R}$ gleichmächtig.

Beweis

Führt man die Relation $f \subseteq \tilde{R} \times O$ gemäß der Beziehung

$$\tilde{\rho} f O_{ij} : \iff \pi_\alpha(\tilde{\rho}) = (a_i, a_j)$$

ein, dann kann man ohne weiteres zeigen, daß f eine bijektive Abbildung von $\tilde{R}$ in O ist.

Nach diesem Satz kann man also zur Behandlung von Prozessen sowohl $\tilde{R}$ als auch O zugrundelegen.

Zur Erläuterung der Arbeitsweise eines Prozesses ist die Einführung des Begriffes "Berechnung" erforderlich. Zuerst wird der Begriff "Prozeßschritt" definiert.

1.1.8 Definition

Ist zu einem gegebenen Zeitpunkt der Zustand q eines Prozesses gleich (a,d) , so ist ein direkter Übergang (Prozeßschritt) zum Zustand $q'=(a',d')$ genau dann möglich, wenn es ein $\tilde{\rho} \in \tilde{R}$ gibt mit $(q,q') \in \tilde{\rho}$.

In Zeichen $(a,d) \xrightarrow{\tilde{\rho}} (a',d')$

Mit Hilfe der im Beweis von Satz 1.1.7 definierten Abbildung f kann man diese Definition auch folgendermaßen formulieren:

1.1.9 Definition

Ist zu einem gegebenen Zeitpunkt der Zustand q_i eines Prozesses gleich (a_i,d_i) , so ist ein direkter Übergang (Prozeßschritt) zum Zustand $q_j=(a_j,d_j)$ genau dann möglich, wenn es ein $O_{ij} \in O$ gibt mit $(d_i,d_j) \in O_{ij}$.

In Zeichen $(a_i,d_i) \xrightarrow{O_{ij}} (a_j,d_j)$.

1.1.10 Definition

Die Sequenz $q_0, q_1, q_2, \dots, q_i, \dots$

heißt Berechnung eines Prozesses, wenn gilt:

$$a) \quad q_0 = (a_0, d_0) \in \{a_0\} \times D_0$$

$$b) \quad \forall i \ (i \geq 0 \longrightarrow \exists \tilde{\rho} \in \tilde{R} \ (q_i \xrightarrow{\tilde{\rho}} q_{i+1}))$$

1.1.11 Definition

Eine Berechnung $q_0, q_1, q_2, \dots, q_n$

heißt endlich, wenn es kein $\tilde{\rho} \in \tilde{R}$ gibt mit $q_n \in \text{Dom}(\tilde{\rho})$.

Bemerkung:

Sowohl der Definitionsbereich einer Relation r als auch der Wertebereich einer Variablen v wird mit $\text{Dom}(r)$ bzw. $\text{Dom}(v)$ bezeichnet. Die Mächtigkeit einer Menge M wird mit $|M|$ bezeichnet.

1.1.12 Definition

Ein Zustand $p \in Q$ heißt erreichbar, wenn es eine Berechnung gibt, in der der Zustand q vorkommt.

Bezeichnet man mit $Q_0 = \{a_0\} \times D_0$, so ist die Menge der erreichbaren Zustände eines Prozesses gegeben durch

$$Q_0 \cup R[Q_0] \cup R^2[Q_0] \cup \dots,$$

d.h. durch die reflexive, transitive Hülle von Q_0 bezüglich der Relation R .

Die bisherigen Definitionen legen eine sequentielle Arbeitsweise von Prozessen fest.

In dieser Arbeit werden nur sequentielle Prozesse betrachtet. Dieser Einschränkung liegt die Vorstellung zugrunde, daß ein Prozeß zu einer gegebenen Zeit nie verschiedene Aktionen (Prozeßschritte) ausführen sollte. Der Tatsache, daß oft verschiedene Aktivitäten gleichzeitig stattfinden können, wird dadurch Rechnung getragen, daß verschiedene kooperierende Prozesse unter gewissen Bedingungen gleichzeitig Aktionen ausführen können.

Nach bisherigen Erfahrungen sind funktionale Modelle (siehe nächste Definition) ausreichend für die formale Behandlung von Betriebssystemen.

1.1.13 Definition

Ein Prozeß heißt funktional genau dann, wenn

$$\forall \tilde{\rho} (\tilde{\rho} \in \tilde{R} \implies \tilde{\rho} \text{ ist eine Abbildung von } Q \text{ in } Q).$$

Man kann auch sagen, daß ein Prozeß genau dann funktional ist, wenn jedes $O_{ij} \in O$ eine Abbildung von D in D ist.

1.1.14 Definition

Ein Prozeß P heißt deterministisch, wenn für alle nicht-leeren Relationen O_{kl} und O_{km} von P gilt:

$$l \neq m \implies \text{Dom } (O_{kl}) \cap \text{Dom } (O_{km}) = \emptyset \text{ (leer)}.$$

In der Literatur werden die Begriffe "sequentiell" und "deterministisch" nicht immer streng unterschieden. Aus den hier angegebenen Definitionen ist folgendes ersichtlich: Die Eigenschaft "sequentiell" bzw. "nichtsequentiell" hängt mit dem Ablauf, d.h. mit dem dynamischen Verhalten eines Prozesses zusammen. Der Begriff "deterministisch" bzw. "nicht-deterministisch" bezieht sich dagegen auf die Beschreibung, d.h. auf eine statische Eigenschaft des Prozesses.

Alle bekannten Probleme aus der Theorie der Betriebssysteme, insbesondere die zur Untersuchung der Kooperation und Synchronisation von Prozessen, können durch Prozesse gelöst werden, die sowohl funktional als auch deterministisch sind (siehe Beispiele im nächsten Abschnitt).

Ein typisches Merkmal der Betriebsprogrammierung und allgemein der Systemprogrammierung ist die Forderung bzw. der Wunsch, daß die einzelnen, in einem System enthaltenen Prozesse, im Gegensatz zu den klassischen Algorithmen, nicht terminieren sollen.

1.1.15 Definition

Mit $IA = \{1, 2, \dots, n\}$ soll die der Adressenmenge $A = \{a_1, a_2, \dots, a_n\}$ eines Prozesses korrespondierende Indexmenge bezeichnet werden. Ein Prozeß heißt zyklisch, wenn für alle $k \in IA$ mindestens ein $l \in IA$ existiert mit $|O_{kl}| \neq 0$.

Ein zyklischer Prozeß besitzt somit keine "Haltebe-fehle". Die Operationen O_{kl} eines funktionalen Prozesses sind im allgemeinen partielle Abbildungen.

Somit ist es möglich, daß während einer Berechnung eines funktionalen, zyklischen Prozesses ein Zustand erreicht wird, von dem aus keine Operationen dieses Prozesses ausgeführt werden können. Die Bedeutung dieser Eigenschaft wird im nächsten Abschnitt offensichtlich. Vorwegnehmend kann man an dieser Stelle sagen, daß die oben erwähnte Eigenschaft sogar eine Notwendigkeit bei der Konstruktion von Betriebssystemen ist. Um den aus der Praxis stammenden Problemen gerecht zu werden, muß bei der Aufstellung von relevanten theoretischen Modellen folgender physikalischer Sachverhalt berücksichtigt werden: Im Rahmen der Kooperation eines Prozesses P mit anderen Prozessen in einem System ist es möglich, daß ein Zustand q erreicht wird, von dem aus Prozeß P keine Aktionen ausführen kann. In diesem Falle muß Prozeß P solange warten, bis der vorliegende Zustand durch die Mitwirkung von anderen Prozessen des Systems so modifiziert wird, daß Prozeß P seine Aktivitäten fortsetzen kann.

Als letztes sei nun bemerkt, daß das hier entwickelte Modell eines Prozesses ein allgemeines Relationsmodell ist.

Die Zustandsmenge D eines Prozesses kann beliebig interpretiert werden. So könnte sie z.B. die freie Halbgruppe Σ^* über einem Alphabet Σ sein oder eine Menge von Warteschlangen, deren Elemente (Warteschlangen) einzelne Worte über einem Alphabet von Namen sind, oder auch die Menge aller Punkte bzw. Vektoren von gewissen Teilräumen des n -dimensionalen ($n \in \mathbb{N}$) Vektorraumes. Für die Prozeßrelation R kann man dann dementsprechend verschiedene Interpretationen zulassen.

1.2 Systeme von Prozessen

Ein Betriebssystem besteht im allgemeinen aus mehreren Prozessen, die durch geeignete Zusammenarbeit in der Lage sind, (bzw. sein müssen) komplexe Tätigkeiten auszuführen. Ein charakteristisches Merkmal der System- und Betriebsprogrammierung ist die gegenseitige Abhängigkeit der einzelnen Prozesse innerhalb eines Systems von Prozessen. Diese Abhängigkeit wird dadurch berücksichtigt, daß der Fortgang von Prozessen an gewissen Stellen vom Wert mancher Variablen abhängt, die durch andere Prozesse desselben Systems verändert werden. Eng mit dieser Eigenschaft hängen gewisse Situationen zusammen, wie z.B. die der Blockierung und Verklemmung, die wiederum einen wesentlichen Teil der folgenden Kapitel bilden. In diesem Abschnitt wird deshalb der Begriff "Prozeßsystem" eingeführt und behandelt.

1.2.1 Vereinbarung

Mit Rücksicht auf Betriebssysteme wird für die Zustandsmenge Q eines Prozesses folgende Form vereinbart:

$$Q = A \times D_1 \times D_2 \times \dots \times D_N .$$

Dabei ist A eine endliche, nichtleere Adressenmenge, N eine natürliche Zahl und für alle $i \in \{1, 2, \dots, N\}$ gilt

$$D_i = \{0, 1, 2, \dots, n_i\} \text{ oder } D_i = \mathbb{N}_0 .$$

Ein Prozeßzustand $q \in Q$ ist dann ein $(N+1)$ -Tupel der Form $q = (a, d_1, d_2, \dots, d_N) = (a, \underline{d})$, wobei $a \in A$

und $\forall i (1 \leq i \leq N \rightarrow d_i \in D_i)$ gilt.

Ein Zustand von P ist also ein $(N+1)$ -Tupel von Werten, d.h. von Größen mit einer festen Bedeutung. Für einen vorgegebenen Prozeß kann man nun eine $(N+1)$ -dimensionale Vektorvariable (Zustandsvariable von P) $\underline{v} = (v_0, v_1, \dots, v_N)$ einführen. Diese Variable darf Werte aus der Zustandsmenge Q des Prozesses nehmen. Dies heißt, der Wertebereich der Variablen v_0 ist A , während der Wertebereich der Variablen v_i ($1 \leq i \leq N$) die Menge D_i ist.

Hat ein Prozeß eine endliche Berechnung $q_0, q_1, \dots, q_i$ durchgeführt, so besitzt die Variable t (auch virtuelle Zeit genannt) den Wert i . Während dieser Berechnung wird der Variablen $\underline{v}$ zum virtuellen Zeitpunkt $t=i$ der Wert q_i zugeordnet.

Die Variable v_0 wird auch Adreßvariable genannt, während die Variable v_i ($1 \leq i \leq N$) als die i -te Datenvariable des Prozesses bezeichnet wird.

Sei nun eine endliche, nichtleere Menge $\bar{P} = \{P_1, P_2, \dots, P_K\}$ von sequentiellen, deterministischen und funktionalen Prozessen gegeben.

Für P_i ($1 \leq i \leq K$) mit

$$P_i = (A_i, \underline{D}_i, R_i, a_{i0}, \underline{D}_{i0})$$

$$\text{sei } \underline{D}_i = D_{i1} \times D_{i2} \times \dots \times D_{iN_i}.$$

Die Zustandsvariable von P_i wird mit

$$\underline{v}_i = (v_{i0}, v_{i1}, \dots, v_{iN_i})$$

bezeichnet.

Für die Wertebereiche A_i der Variablen v_{i0} gelte:

- 1) $\forall i (1 \leq i \leq K \rightarrow A_i = (a_{i0}, a_{i1}, \dots, a_{iZ_i})$ ist eine nichtleere, endliche Menge), (*)
- 2) $\forall i \forall j (1 \leq i, j \leq K \wedge i \neq j \rightarrow A_i \cap A_j = \emptyset)$.

Mit I_L ($1 \leq L \leq K$) wird weiterhin die Menge

$$\{(L, 1), (L, 2), \dots, (L, N_L)\}$$

bezeichnet.

Im folgenden wird ein Prozeß $P = (\underline{A}, \underline{D}, R, \underline{a}_0, \underline{D}_0)$ konstruiert, der das Verhalten des aus $\bar{P}$ bestehenden Systems beschreibt. Prozeß P wird deswegen ein aus $\bar{P}$ zusammengesetztes Prozeßsystem genannt.

Für $\underline{A}$ und $\underline{a}_0$ müssen folgende Beziehungen gelten:

$$\underline{A} = A_1 \times A_2 \times \dots \times A_K,$$

$$\underline{a}_0 = (a_{10}, a_{20}, \dots, a_{K0})$$

Für die Datenmenge $\underline{D}$ von P gelte: $\underline{D} = \tilde{D}_1 \times \tilde{D}_2 \times \dots \times \tilde{D}_M$.

Für die Zustandsvariable $\underline{v}$ von P kann man dann schreiben:

$$\underline{v} = (\tilde{v}_0, \tilde{v}_1, \dots, \tilde{v}_M)$$

Die Abhängigkeit der einzelnen Prozesse von $\bar{P}$ untereinander wird erreicht, indem eine Variable eines Prozesses $P_i \in \bar{P}$ mit einer Variablen eines anderen Prozesses

$P_j \in \bar{P}$ identifiziert wird. Dies wird mit Hilfe von Id

(siehe unten) realisiert.

1.2.2 Definition

Eine totale Abbildung Id ,

$$\text{Id} : \{(i,j) \mid 1 \leq i \leq K \wedge 1 \leq j \leq N_i\} \longrightarrow \{1,2,\dots,M\} ,$$

heißt Variablenidentifikation, wenn

- 1) für alle L ($1 \leq L \leq K$) die Einschränkung von Id auf I_L eine injektive Abbildung ist,
- 2) für alle Paare (i,j) aus dem Definitionsbereich von Id gilt:

$$\text{Id}(i,j) = 1 \Rightarrow \text{Dom}(v_{ij}) = \text{Dom}(\tilde{v}_1)$$

- 3) Id surjektiv ist.

Aus 1) folgt, daß verschiedene Variablen eines Prozesses nicht identifiziert werden dürfen.

Aus Bedingung 2) folgt, daß für alle Paare (k,l) und (m,n) aus dem Definitionsbereich von f_{Id} gilt:

$$\text{Id}(k,l) = \text{Id}(m,n) \Rightarrow \text{Dom}(v_{kl}) = \text{Dom}(v_{mn}) .$$

Das Prozeßsystem P darf laut Bedingung 3) keine "überflüssigen" Variablen enthalten.

Für ein vorgegebenes System $\bar{P} = \{P_1, P_2, \dots, P_K\}$ hat man im allgemeinen mehrere Möglichkeiten, die Datenmenge $\underline{D}$ von P zu konstruieren und zwar je nach Wahl der oben erwähnten Identifikationsfunktion Id .

Für die Zustandsmenge Q von P gilt:

$$Q = \underline{A} \times \underline{D}$$

Für ein $q \in Q$ schreibt man auch

$$q = (\underline{a}, \tilde{d}_1, \tilde{d}_2, \dots, \tilde{d}_M),$$

wobei $\underline{a}$ ein festes K -Tupel von Adressen ist, d.h.

$$\underline{a} = (a_{11_1}, a_{21_2}, \dots, a_{K1_K}).$$

Dabei ist für alle $i (1 \leq i \leq M)$ $\tilde{d}_i \in \tilde{D}_i$ und alle $j (1 \leq j \leq K)$ $a_{j1_j} \in A_j$.

1.2.3 Definition

Die Adresse des Prozesses P_i im oben erwähnten Zustand q wird mit $\pi\alpha(q, i)$ bezeichnet, d.h.

$$\pi\alpha(q, i) = a_{i1_i} \quad \text{für } 1 \leq i \leq K.$$

Für den Prozeß P_i wird das N_i -Tupel $\pi\delta(q, i)$ folgendermaßen definiert:

$$\pi\delta(q, i) = (\tilde{d}_{Id(i, 1)}, \tilde{d}_{Id(i, 2)}, \dots, \tilde{d}_{Id(i, N_i)})$$

Im Zustand $q \in Q$ von P ist dann der momentane Zustand von P_i gegeben durch $\psi(q, i)$, wobei

$$\psi(q, i) = (\pi\alpha(q, i), \pi\delta(q, i))$$

Von der Menge $\underline{D}_0 \subseteq \underline{D}$ von P wird weiterhin gefordert, daß für alle i ($1 \leq i \leq K$) und alle

$$q_0 = (\underline{a}_0, \underline{d}_0) \in \{ \underline{a}_0 \} \times \underline{D}_0 \quad \text{gilt:}$$

$$\psi(q_0, i) \in \{ \underline{a}_{i0} \} \times \underline{D}_{i0}$$

Zur vollständigen Beschreibung des aus $\bar{P}$ zusammengesetzten Prozesses P ist noch die Angabe der Relation R von P erforderlich. Zuvor werden aber einige Vorbemerkungen gemacht. Beschränkt man sich auf solche Systeme, bei denen zu jeder Zeit höchstens ein Prozeß eine Aktion ausführt, dann ist die Angabe von R relativ einfach (siehe [53]). Für die Modellierung realer Systeme aber ist erwünscht, daß mehrere Prozesse gleichzeitig gewisse Aktionen ausführen können, falls sie "verträglich" sind. Für reale Systeme dauern außerdem die Aktionen von Prozessen eine gewisse Zeit, die aus verschiedenen, nicht immer mit der gerade stattfindenden Aktion zusammenhängenden Gründen länger oder kürzer sein kann. Um ein möglichst wirklichkeitsnahes Modell zu entwickeln, werden auch diese Gesichtspunkte bei der Definition von R zu berücksichtigen sein.

Für einen Prozeß P_i aus $\bar{P}$ werden die Funktionen O_{kl}^i nach Definition 1.1.6 und die Mengen O^i nach Definition 1.1.7 eingeführt.

Jede Operation O_{kl}^i eines Prozesses P_i aus $\bar{P}$ ist eine im allgemeinen partielle Abbildung von $\underline{D}_i$ in sich;

somit ist sie (aufgrund der Identifikationsfunktion Id) auch eine Abbildung von

$$\tilde{D}_{Id(i,1)} \times \tilde{D}_{Id(i,2)} \times \dots \times \tilde{D}_{Id(i,N_i)} \quad \text{in sich.}$$

Wird also eine Operation O_{kl}^i auf der Datenbasis $\tilde{D}_1 \times \tilde{D}_2 \times \dots \times \tilde{D}_M$ des Prozeßsystems P ausgeführt, so hat dies die selbe Wirkung wie die Ausführung dieser Operation auf der Datenbasis

$$\begin{aligned} \underline{D}_i = D_{i1} \times D_{i2} \times \dots \times D_{iN_i} &= \tilde{D}_{Id(i,1)} \times \tilde{D}_{Id(i,2)} \times \dots \\ &\dots \times \tilde{D}_{Id(i,N_i)} \end{aligned}$$

des Prozesses P_i .

Ein Prozeß $P_i \in \bar{P}$ kann also höchstens die Variablen aus

$$\left\{ \tilde{v}_{Id(i,1)}, \dots, \tilde{v}_{Id(i,N_i)} \right\} \subseteq \left\{ \tilde{v}_1, \tilde{v}_2, \dots, \tilde{v}_M \right\} \quad \text{des}$$

Prozeßsystems P abfragen bzw. manipulieren. Die Menge der Variablen aus $\left\{ \tilde{v}_1, \tilde{v}_2, \dots, \tilde{v}_M \right\}$, deren Wert entscheidend für die Ausführbarkeit von O_{kl}^i ist, wird Menge der Eingabevariablen von O_{kl}^i genannt und mit $EV(O_{kl}^i)$ bezeichnet. Die Menge $AV(O_{kl}^i)$ der Ausgabevariablen von O_{kl}^i ist die Menge der Variablen aus $\left\{ \tilde{v}_1, \tilde{v}_2, \dots, \tilde{v}_M \right\}$, die bei einer Ausführung von O_{kl}^i manipuliert werden. Der Schnitt von $EV(O_{kl}^i)$ und $AV(O_{kl}^i)$ muß nicht leer sein. Die Vereinigung dieser Mengen kann auch eine echte Teilmenge von

$$\left\{ \tilde{v}_{Id(i,1)}, \dots, \tilde{v}_{Id(i,N_i)} \right\} \quad \text{sein.}$$

Über der Menge $O = O^1 \cup O^2 \cup \dots \cup O^K$ wird folgende Relation eingeführt:

1.2.4 Definition

Unter der Verträglichkeitsrelation V versteht man folgende Menge:

$$\begin{aligned} V &= \left\{ (O_{kl}^i, O_{mn}^j) \mid i \neq j \wedge P_i, P_j \in \bar{P} \wedge EV(O_{kl}^i) \cap AV(O_{mn}^j) = \right. \\ &= \left. AV(O_{kl}^i) \cap EV(O_{mn}^j) = AV(O_{kl}^i) \cap AV(O_{mn}^j) = \emptyset \right\} \end{aligned}$$

1.2.5 Korollar

Die Verträglichkeitsrelation ist symmetrisch und irre-

flexiv (d.h. $\forall O_{kl}^i \in Q (O_{kl}^i, O_{kl}^i) \notin V$), braucht aber nicht transitiv zu sein.

1.2.6 Definition

Ein Prozeß $P_i \in \bar{P}$ heißt lauffähig bezüglich $q \in Q$, wenn gilt

$$\pi\alpha(q, i) = a_{i_k} \Rightarrow \exists O_{kl}^i \quad (\pi\delta(q, i) \in \text{Dom}(O_{kl}^i))$$

Da hier nur deterministische Prozesse betrachtet werden, kann es in der oberen Beziehung höchstens eine Funktion

O_{kl}^i geben, die in dem Zustand q definiert ist.

1.2.7 Definition

Zwei Prozesse $P_i, P_j \in \bar{P}$ ($i \neq j$) heißen q -verträglich, wenn beide Prozesse bgl. q ($q \in Q$) lauffähig sind und für die aus Definition 1.2.5 eindeutig zu bestimmenden Funktionen O_{kl}^i und O_{mn}^j gilt $(O_{kl}^i, O_{mn}^j) \in V$.

1.2.8 Satz

Sind die Prozesse P_k, P_l und P_m paarweise q -verträglich und bezeichnet man mit $f(P_k P_l P_m, q)$ den von q aus erreichten Zustand, wenn die Prozesse P_k, P_l und P_m in dieser Reihenfolge ihre Aktionen beenden, so gilt:

$$a) \quad f(P_k P_l, q) = f(P_l, f(P_k, q)) = f(P_k, f(P_l, q)) =$$

$$= f(P_l P_k, q) \quad (\text{Kommutativität})$$

$$b) \quad f(P_m, f(P_k P_l, q)) = f(P_l P_m, f(P_k, q))$$

(Assoziativität)

Beweis

Da die Prozesse P_k, P_l und P_m paarweise q -verträglich sind, ist die Aktionswirkung eines dieser Prozesse unabhängig von der Aktionswirkung eines anderen dieser Prozesse. Weiterhin werden hier nur funktionale und deterministische Prozesse betrachtet, sodaß die oberen Beziehungen offensichtlich sind.

Nun kann man die Relation R des Prozeßsystems definieren.

1.2.9 Definition

Für zwei Zustände $q, q' \in Q$ (Q ist die Zustandsmenge von P) gilt $(q, q') \in R$, genau dann, wenn es eine Menge

$\{P_1, P_2, \dots, P_k\} \subseteq \bar{P}$ von paarweise q -verträglichen Prozessen gibt und $f(P_1 P_2 \dots P_k, q) = q'$ gilt.

Zusammenfassend wird jetzt der Begriff "Prozeßsystem" nach folgender Definition eingeführt.

1.2.10 Definition

Sei $\bar{P} = \{P_1, P_2, \dots, P_K\}$ eine Menge von sequentiellen Prozessen mit $P_i = (A_i, \underline{D}_i, R_i, a_{i0}, \underline{D}_{i0})$ und $\underline{D}_i = D_{i1} \times D_{i2} \times \dots \times D_{iN_i}$ für $1 \leq i \leq K$. Für A_i gelten die Beziehungen (*) aus Seite 21. Ein nicht unbedingt sequentieller Prozeß $P = (\underline{A}, \underline{D}, R, \underline{a}_0, \underline{D}_0)$ heißt genau dann ein aus $P_1, P_2, \dots, P_K$ zusammengesetztes Prozeßsystem, wenn gilt:

- 1) $\underline{A} = A_1 \times A_2 \times \dots \times A_K$
- 2) $\underline{a}_0 = (a_{10}, a_{20}, \dots, a_{K0})$
- 3) Ist $\underline{D} = \tilde{D}_1 \times \tilde{D}_2 \times \dots \times \tilde{D}_M$, so gibt es eine Funktion Id zur Variablenidentifikation nach Definition 1.2.2
- 4) Für alle $q_0 = (\underline{a}_0, \underline{d}_0) \in \{\underline{a}_0\} \times \underline{D}_0$ und alle i mit $P_i \in \bar{P}$ gilt: $\psi(q_0, i) \in \{a_{i0}\} \times \underline{D}_{i0}$
- 5) R wird nach Definition 1.2.9 konstruiert.

Bemerkung 1

Obwohl die einzelnen Prozesse des zugrundegelegten Prozeßsystems deterministisch und funktional sind, ist R im allgemeinen keine Funktion, sondern eine Relation. Dies

rührt von der Tatsache her, daß von einem Zustand q aus verschiedene Prozesse ihre Aktionen ausführen können.

1.2.11 Definition

Eine Menge lp von Prozessen ($lp \subseteq \bar{P}$) heißt q -verträglich, wenn alle Paare (P_i, P_j) mit $P_i, P_j \in lp$ und $i \neq j$ q -verträglich sind. Zwei Mengen lp und lfp von Prozessen ($lp, lfp \in \bar{P}$) mit $lp \cap lfp = \emptyset$ heißen schließlich q -verträglich, wenn alle Paare (P_i, P_j) mit $i \neq j$ und $P_i, P_j \in lp \cup lfp$ q -verträglich sind.

Die Werte der Variablen lp , lfp und $\underline{v}$ (Zustandsvariable des Prozeßsystems P) zum virtuellen Zeitpunkt t werden mit $lp(t)$, $lfp(t)$ und $\underline{v}(t)$ bezeichnet. In der folgenden Definition 1.2.12 wird $lp(t)$ sämtliche Prozesse enthalten, die zum Zeitpunkt t einen Prozessor zugeteilt bekommen haben und gerade eine Operation ausführen. Weiterhin wird in derselben Definition $lfp(t)$ so konstruiert, daß $lp(t)$ und $lfp(t)$ im Sinne der Definition 1.2.11 q -verträglich sind, wobei $q = \underline{v}(t)$ ist. Gilt $P_i \in lp(t)$, so sagt man auch, Prozeß P_i befinde sich zum Zeitpunkt t im Zustand "laufend". Gilt $P_i \in lfp(t)$, so sagt man auch, Prozeß P_i befinde sich zum Zeitpunkt t im Zustand "laufbereit". Jeder laufbereite Prozeß ist auch (im Sinne der Definition 1.2.6) lauffähig. In der Definition 1.2.12 wird sich aber herausstellen, daß nicht jeder lauffähige Prozeß in den Zustand "laufbereit" versetzt werden kann. Durch das Tripel $(\underline{v}(t), lp(t), lfp(t))$ wird der erweiterte Zustand des Prozeßsystems P zum virtuellen Zeitpunkt t charakterisiert. In der nächsten Definition wird die Arbeitsweise des zusammengesetzten Prozeßsystems beschrieben.

1.2.12 Definition

- a) Zum virtuellen Zeitpunkt $t = 0$ (unmittelbar nach der Initialisierung des Systems) sei

$$1) \quad \underline{v}(0) = q_0 \in \{ \underline{a}_0 \} \times \underline{D}_0$$

$$2) \quad lp(0) = \emptyset \text{ (leer) und}$$

$$3) \quad lfp(0) \text{ irgendeine } q_0\text{-verträgliche Teilmenge von } \bar{P}.$$

Dann ist folgender Schritt des Prozeßsystems P möglich:

$$(\underline{v}(0), lp(0), lfp(0)) \xrightarrow{P} (\underline{v}(1), lp(1), lfp(1)),$$

wobei

$$4) \quad \underline{v}(1) = \underline{v}(0) = q_0,$$

$$5) \quad lp(1) \subseteq lfp(0)$$

$$6) \quad lfp(1) = lfp(0) - lp(1)$$

Bei diesem Schritt bleibt der Wert von $\underline{v}$ unverändert. Dagegen wandern manche lafbereite Prozesse in die Menge der laufenden Prozesse. Die Mengen $lp(1)$ und $lfp(1)$ sind offensichtlich q_0 -verträglich.

- b) Der erweiterte Zustand von P sei zum virtuellen Zeitpunkt t charakterisiert durch $(\underline{v}(t), lp(t), lfp(t))$, wobei $\underline{v}(t) = q \in Q$ ist und die Mengen $lp(t)$ und $lfp(t)$ q -verträglich sind. Dann sind zwei Schritttarten von P möglich.

$$b1) \quad (\underline{v}(t), lp(t), lfp(t)) \xrightarrow{P} (\underline{v}(t+1), lp(t+1), lfp(t+1)),$$

wobei

- 1) $\underline{v}(t+1) = \underline{v}(t) = q,$
- 2) $lfp(t+1) = lfp(t) - lfp',$ mit $lfp' \subseteq lfp(t),$
- 3) $lp(t+1) = lp(t) + lfp'$

Wie im Fall a) wird hier zum Zeitpunkt $t+1$ eine Menge lfp' von Prozessen vom Zustand "laufbereit" in den Zustand "laufend" versetzt. Die Mengen $lp(t+1)$ und $lfp(t+1)$ sind offensichtlich q -verträglich.

b2) Einige laufende Prozesse beenden ihre Aktionen:

$$(\underline{v}(t), lp(t), lfp(t)) \xrightarrow{P} (\underline{v}(t+1), lp(t+1), lfp(t+1)).$$

Ist $lp' \subseteq lp(t)$ die Menge der Prozesse, die zum Zeitpunkt $t+1$ ihre Aktionen beenden, so gilt:

- 4) $lp(t+1) = lp(t) - lp'$
- 5) Aus $q = \underline{v}(t)$ und lp' ist $q' = \underline{v}(t+1)$ eindeutig bestimmt (siehe Satz 1.2.8). Es gilt $(q, q') \in R.$
- 6) Zur Bildung von $lfp(t+1)$ hat man zwei Möglichkeiten:

$$\alpha) lfp(t+1) = lfp(t) + lfp', \text{ wobei } lfp' \subseteq \bar{P} - lp(t+1)$$

Die Menge der zum Zeitpunkt $t+1$ laufbereiten Prozesse enthält alle zum Zeitpunkt t laufbereiten Prozesse. Sie kann weitere nichtlaufende Prozesse (lfp') aufnehmen, solange sie mit $lp(t+1)$ q' -verträglich bleibt.

- β) Zum Zeitpunkt $t+1$ kann aber die Mengenvariable lfp auch einen Wert annehmen, der keine Obermenge von $lfp(t)$ ist (siehe Bemerkung 2). Auf jeden Fall müssen $lp(t+1)$ und $lfp(t+1)$ q' -verträglich sein.

Bemerkung 2

Sind in einem Prozeßsystem P die Mengen $lp(o)$ und $lfp(o)$ q_0 -verträglich und sind in P nur Schritte nach Definition 1.2.12 erlaubt, so gilt zu jedem Zeitpunkt t mit $\underline{v}(t)=q$: $lp(t)$ und $lfp(t)$ sind q -verträglich.

Zum Falle β) von b2) in Definition 1.2.12 ist nun folgendes zu sagen: Die Verträglichkeitsrelation nach Definition 1.2.7 ist zwar symmetrisch aber nicht transitiv. So ist z.B. möglich, daß sowohl P_1 und P_2 als auch P_2 und P_3 , nicht aber P_1 und P_3 q -verträglich sind. Ist zu einem virtuellen Zeitpunkt t $\underline{v}(t)=q$ und ist sowohl $\{P_1, P_2\}$ als auch $\{P_2, P_3\}$ mit $lp(t)$ q -verträglich, so könnte $lfp(t)$ den Wert $\{P_1, P_2\}$ oder $\{P_2, P_3\}$ nicht aber $\{P_1, P_2, P_3\}$ annehmen. Ist nun $lfp(t) = \{P_1, P_2\}$, so ist nach β) in b2) möglich, daß $lfp(t+1) = \{P_2, P_3\}$ gilt, obwohl $P_1 \in lp(t+1)$ ist. In diesem Falle kann also $lfp(t+1)$ einen Wert annehmen, der keine Obermenge von $lfp(t)$ ist, unter der Voraussetzung natürlich, daß $lfp(t+1)$ mit $lp(t+1)$ q' -verträglich ist.

Dieses Modell läßt also einen freien Spielraum dafür, welche Werte die Mengenvariable lfp nach Ausführung von Schritt b2 annimmt. Diese Freizügigkeit hat einen praktischen Hintergrund: Man kann dem Modell problemspezifische Stra-

tegien für die Auswahl der lfp-Werte bei den entsprechenden Schritten überlagern. Auch vor Ausführung des Schrittes a) kann man im allgemeinen der Variablen lfp verschiedene Werte zuordnen.

Das vorliegende Modell erlaubt eine parallele Ausführung von mehreren sequentiellen Prozessen. Der spezielle Fall des sequentiellen Prozesses ist im allgemeinen Fall enthalten und ist dadurch charakterisiert, daß der Adreßteil des Prozeßsystems nur aus einer Komponenten besteht und die Menge $lp(lfp)$ entweder leer (einelementig) oder einelementig (leer) ist.

Somit wurde ein einheitliches Modell sowohl für einfache als auch für zusammengesetzte Prozesse entwickelt.

Ähnlich wie bei den einfachen Prozessen kann man den Begriff der Berechnung auch bei zusammengesetzten Prozessen einführen.

1.2.13 Definition

Eine Sequenz $q_0, q_1, q_2, \dots, q_i, \dots$ heißt Berechnung eines Prozeßsystems, wenn

$$a) \quad q_0 \in \{ \underline{a_0} \} \times \underline{D_0}$$

$$b) \quad \text{für alle } i \geq 0 \text{ gilt: Es gibt einen Schritt nach b2 (Def. 1.2.12) mit } q_i = \underline{v}(t) \text{ und } q_{i+1} = \underline{v}(t+1).$$

Eine endliche Berechnung kann analog zu Definition 1.1.10 eingeführt werden. Ist zu einem virtuellen Zeitpunkt t der Gesamtzustand des Systems gleich $(q(t), lp(t), lfp(t))$, so ist aus dem bereits geschilderten offensichtlich, daß alle Prozesse aus $lp(t)$ mit Sicherheit ihre Aktionen beenden werden. Das Endergebnis nach Beendigung der Aktionen aller in $lp(t)$ enthaltenen Prozesse ist sogar unabhängig von der Reihenfolge, in der die oben genannten Aktionen beendet

werden. Interessiert man sich nur für die aus einem Anfangszustand q_0 erreichbaren Zustände, so kann man ohne weiteres davon ausgehen, daß jede Aktion, die zu einem Zeitpunkt t startet, auch zum selben Zeitpunkt beendet wird. In diesem Fall braucht man also nicht zwischen Starten und Beenden einer Aktion zu unterscheiden. Wichtig ist allein die Tatsache, daß eine Aktion stattgefunden hat bzw. nicht.

Die im Satz 1.2.8 eingeführte Funktion f ist laut Definition immer definiert, wenn man vereinbart, daß $f(\Lambda, q) = q$ gilt, wenn Λ die leere Sequenz von Prozessen ist.

In der nächsten Definition wird eine Verallgemeinerung dieser Funktion gegeben.

1.2.14 Definition

Sei $\bar{P} = \{P_1, P_2, \dots, P_m\}$ die Menge der sequentiellen Prozesse eines Prozeßsystems P . Dann ist $\bar{P}^*$ die Menge aller endlichen Sequenzen mit Elementen aus $\bar{P}$ einschließlich der leeren Sequenz. Die totale Abbildung $\tilde{f}$

$$\tilde{f}: \bar{P}^* \rightarrow Q$$

wird nach folgender rekursiven Definition eingeführt:

- a) $\tilde{f}(\Lambda) = q_0$
- b) Sei $s = P_{i_1} P_{i_2} \dots P_{i_K}$ eine Sequenz aus $\bar{P}^*$ mit

$$\tilde{f}(s) = q. \text{ Ist } s' = s P_{i_{K+1}} \in \bar{P}^* \text{ und } i_{K+1} = 1$$

so ist

$$\tilde{f}(sP_1) = \begin{cases} O_{mn}^1(q) & \text{Prozeß } P_1 \text{ bgl. } q \text{ lauf-} \\ & \text{fähig ist} \\ q & \text{falls} \\ & \text{sonst} \end{cases}$$

Dabei ist laut Definition 1.2.6 die Funktion O_{mn}^1 eindeutig bestimmt.

Betrachtet man eine einelementige Sequenz P_1 , so sieht man folgenden Unterschied zwischen dieser Sequenz und der Operation O_{OK}^i :

Ist $q_0 \in \text{Dom}(O_{OK}^i)$, so stimmen $O_{OK}^i(q_0)$ und $\tilde{f}(P_1)$

überein, anderenfalls ist $O_{OK}^i(q_0)$ undefiniert, während $\tilde{f}(P_1)$ definiert und zwar gleich q_0 ist. Hinter dieser Konstruktion steckt folgender physikalischer Sachverhalt: Bei der Realisierung von elementaren Synchronisations-Verfahren muß man den Fall berücksichtigen, daß ein Prozeß an manchen Stellen "stehen bleibt" und auf die Erfüllung einer gewissen Bedingung wartet. Dieses "Stehenbleiben" kann man auf zwei verschiedene Weisen realisieren. 1) Dem Prozeß wird der Prozessor entzogen und ihm erst dann wieder zugeteilt, wenn die oben genannte Bedingung erfüllt ist. In diesem Fall sagt man, der Prozeß sei zuerst in den Zustand "blockiert" und dann wieder in den Zustand "aktiv" versetzt worden. Dieser Fall würde hier dadurch berücksichtigt, daß die einzelnen Operationen eines Prozesses im allgemeinen partielle Funktionen sind. 2) Das "Stehenbleiben" kann man aber auch dadurch realisieren, daß der Prozeß den Prozessor auch dann beibehält, wenn die oben genannte Bedingung nicht erfüllt ist. In diesem Fall darf natürlich der Pro-

zeß keine Änderung des momentanen Zustandes bewirken.
Dies kann man durch Operation der Art

M: if B then S else goto M

modellieren. Dabei ist M eine Adresse, B eine Bedingung und S eine Wertzuweisung an q. Diese Art der Realisierung nennt man "aktives Warten".

Die Art und Weise der Implementierung bzw. Modellierung des "Stehenbleibens" hängt weitgehend von den Zielen und Zwecken des Programms bzw. des Modells ab. Bei der Implementierung muß man natürlich besondere Rücksicht auf die Hardware der zur Verfügung stehenden Anlage nehmen. In dem hier entwickelten Modell wurde für die einzelnen Prozesse das "passive Warten" modelliert, indem nämlich die einzelnen Aktionen O_{kl}^i des einfachen Prozesses P_i im allgemeinen als partielle Funktionen eingeführt wurden. Dagegen wurde für zusammengesetzte Prozesse durch die Einführung der totalen Funktion $\tilde{f}$ (Definition 1.2.14) das "aktive Warten" modelliert. Der Grund für diese Einteilung liegt nicht nur in der Absicht, beide Darstellungsmöglichkeiten zu demonstrieren, sondern in erster Linie in der folgenden Überlegung: Einzelne Prozesse eines Systems sollte man bei Bedarf verdrängen ("passives Warten"), während das ganze System, als eine Einheit betrachtet, nie verdrängt werden sollte. Dies gilt natürlich unter der Voraussetzung, daß das betrachtete System korrekt ist. Es sei nochmals betont, daß in der System- und Betriebsprogrammierung im Gegensatz zu der üblichen Programmierung die Terminierung nicht erwünscht und nicht beabsichtigt ist.

Sowohl implementierungstechnisch als auch theoretisch hat jede Art der Auffassung des "Stehenbleibens" ihre Konsequenzen. Über die theoretischen Konsequenzen der beiden

Auslegungen wurde bereits gesprochen. Aus der Praxis ist wiederum bekannt, daß das "aktive Warten" einen geringen organisatorischen Aufwand, dafür aber eine uneffektive Ausnutzung der Prozessoren mit sich bringt. Für das "passive Warten" gilt gerade das Gegenteil.

Im folgenden werden einige aus der Literatur bekannte Eigenschaften von Systemen mit Hilfe des hier entwickelten Modells eingeführt und behandelt.

Die Funktion $\tilde{f}$ (Definition 1.2.14) wurde unter Zugrundelegung eines festen Anfangszustandes q_0 eingeführt. Wird nun statt q_0 irgendein beliebiger Zustand $q \in Q$ zugrunde gelegt, so wird dies durch die Schreibweise

$$\tilde{f}(s, q)$$

berücksichtigt, wobei $s \in \bar{P}^*$ und $q \in Q$ ist. Auf die formale Einführung von $\tilde{f}(s, q)$ wird hier verzichtet, da sie aus dem bereits Erwähnten selbstverständlich ist. Da im folgenden nur $\tilde{f}(s, q)$ benutzt wird und somit keine Verwechslungsgefahr mit der Funktion $f(s, q)$ vom Satz 1.2.8 besteht, wird statt $\tilde{f}(s, q)$ einfach $f(s, q)$ geschrieben.

1.2.15 Definition

Ein Prozeßsystem P heißt halbkommutativ, wenn es folgende Eigenschaft besitzt:

$$\begin{aligned} \forall q \forall P_i \forall P_j ([q \in Q \wedge P_i, P_j \in \bar{P} \wedge i \neq j \wedge f(P_i, q) \neq q \wedge f(P_j, q) \neq q \wedge \\ \wedge f(P_i P_j, q) \neq f(P_i, q) \wedge f(P_j P_i, q) \neq f(P_j, q)] \implies \\ f(P_i P_j, q) = f(P_j P_i, q)) \end{aligned}$$

1.2.16 Definition

Ein Prozeßsystem P heißt kommutativ, wenn gilt:

$$\forall q \forall P_i \forall P_j (q \in Q \wedge P_i, P_j \in \bar{P} \wedge i \neq j \Rightarrow f(P_i P_j, q) = f(P_j P_i, q))$$

1.2.17 Satz

Es gibt halbkommulative Systeme, die nicht kommutativ sind.

Beweis

Sei P ein halbkommutatives System, $q \in Q$ und $P_i, P_j \in \bar{P}$ mit $i \neq j$. Weiterhin sei

$$\begin{array}{ll} \text{a) } f(P_i, q) = q' \neq q & \text{b) } f(P_j, q') = q' \\ \text{c) } f(P_j, q) = q'' \neq q & \text{d) } f(P_i, q'') = q'' \quad \text{e) } q' \neq q'' \end{array}$$

Dann gilt

$$f(P_i P_j, q) = f(P_j, f(P_i, q)) = f(P_j, q') = q' = f(P_i, q)$$

Andererseits aber ist

$$f(P_j P_i, q) = f(P_i, f(P_j, q)) = f(P_i, q'') = q'' = f(P_j, q)$$

$$\text{Es ist also } f(P_i P_j, q) \neq f(P_j P_i, q)$$

1.2.18 Satz

Jedes kommutative Prozeßsysteme ist halbkommutativ.

Beweis

Trivial (siehe Definitionen 1.2.15 und 1.2.16).

Betriebssysteme, die Nachrichtenpuffer und Hardware-Einheiten verwalten sind typische Beispiele für halb-kommutative Systeme. Die Operationen "Belegen von freien Geräten" bzw. "Vernichten von Nachrichten" und "Freigeben von Geräten" bzw. "Erzeugen von Nachrichten" sind offensichtlich halbkommutativ. Solche Systeme lassen sich mit Hilfe von halbkommutativen Modellen formal darstellen und behandeln. Die rein kommutativen Systeme sind für die Praxis der Betriebssysteme von geringerer Bedeutung.

Persistente Systeme (siehe nächste Definition) weisen folgende Eigenschaft auf: Können zwei Prozesse P_i und P_j den momentanen Zustand q verändern, so kann auch P_i (bzw. P_j) den Zustand $f(P_j, q)$ (bzw. $f(P_i, q)$) verändern.

1.2.19 Definition

Ein Prozeßsystem heißt persistent, wenn gilt:

$$\forall q \forall P_i \forall P_j ([q \in Q \wedge P_i, P_j \in \bar{P} \wedge \neg] \wedge f(P_i, q) \neq q \wedge f(P_j, q) \neq q] \Rightarrow \\ \Rightarrow f(P_i, P_j, q) \neq f(P_i, q))$$

Die Synchronisationsgraphen sind ein bekanntes Beispiel für persistente Systeme, die eine relativ einfache formale Behandlung zulassen. Der für Betriebssysteme elementare "gegenseitige Ausschluß" kann leider nicht mit Hilfe von persistenten Systemen dargestellt werden. Somit sind solche Systeme für die Praxis der Betriebsprogrammierung uninteressant. Diese Eigenschaft wurde hier deswegen erwähnt, weil sie in einer engen Beziehung zur Konfluenz-Eigenschaft (siehe nächste Definition) steht, die auch unter dem Namen "Church-Rosser"-Eigenschaft bekannt ist.

Der Einfachheit wegen werden in den nächsten Beziehungen die Aussagen $q \in Q$ und $P_i \in \bar{P}$ weggelassen.

1.2.20 Definition

Ein Prozeßsystem P heißt lokal konfluent, wenn gilt:

$$\forall q \forall q_1 \forall q_2 \forall P_1 \forall P_j ([f(P_1, q) = q_1 \wedge f(P_j, q) = q_2 \wedge q_1 \neq q_2] \Rightarrow \\ \exists q' \exists P_k \exists P_l (f(P_k, q_1) = q' = f(P_l, q_2)))$$

In der nächsten Definition werden Sequenzen von Prozessen mit s_i bzw. t_i bezeichnet.

1.2.21 Definition

Ein Prozeßsystem heißt global konfluent, wenn gilt:

$$\forall q \forall q_1 \forall q_2 \forall s_1 \forall s_2 ([f(s_1, q) = q_1 \wedge f(s_2, q) = q_2 \wedge q_1 \neq q_2] \Rightarrow \\ \exists q' \exists s'_1 \exists s'_2 (f(s'_1, q_1) = q' = f(s'_2, q_2)))$$

Diese Eigenschaft ist sowohl für die allgemeine Systemprogrammierung als auch für Betriebssysteme von besonderem Interesse. Von den Algorithmen in diesen Bereichen wird nämlich erwartet, daß sie einerseits nie terminieren, andererseits aber wird hier erwünscht, daß sie immer wieder in gewisse (meistens Anfangs-) Zustände zurückkehren können. Global konfluente Systeme weisen diese Eigenschaft auf, wenn in der Beziehung der oberen Definition neben q auch q' ein Anfangszustand ist.

1.2.22 Satz

Die lokale Konfluenz ist eine hinreichende Bedingung für die globale Konfluenz.

Beweis

Für ein lokal konfluentes Prozeßsystem P sei q ein beliebiger Zustand aus Q und s_1, s_2 beliebige Sequenzen von Prozessen aus $\bar{P}$ (d.h. $s_1, s_2 \in \bar{P}^*$), so daß gilt:

$$f(s_1, q) = q_1 \wedge f(s_2, q) = q_2 \wedge q_1 \neq q_2 \quad (1)$$

Der Beweis wird dadurch erbracht, indem hier gezeigt wird, daß aus Beziehung (1) die Gültigkeit der rechten Seite der Implikation aus Definition 1.2.21 folgt.

Sei $s_1 = P_{i_1} P_{i_2} \dots P_{i_K}$. Da es sich in diesem Beweis nur

darum handelt, ob gewisse Zustände erreichbar sind oder nicht, ist es keine Beschränkung der Allgemeinheit, wenn man folgende Annahme macht: Jeder in einer Sequenz s_i aufgerufene Prozeß P_{i_K} ist bezüglich des momentan vorliegenden Zustandes lauffähig im Sinne der Definition 1.2.6. Anderenfalls bewirkt der aufgerufene Prozeß keine Zustandsänderung, sodaß er ohne weiteres aus der Sequenz gestrichen werden kann. Die Länge der Sequenz $s = P_{i_1} P_{i_2} \dots P_{i_K}$ wird mit $|s| = k$ bezeichnet. Statt der Beziehung der Definition 1.2.21 wird folgende Beziehung bewiesen:

$$\begin{aligned} & \forall q \forall q_1 \forall q_2 \forall s_1 \forall s_2 \left([f(s_1, q) = q_1 \wedge f(s_2, q) = q_2 \wedge q_1 \neq q_2] \Rightarrow \right. \\ & \left. \Rightarrow \exists q' \exists s'_1 \exists s'_2 \left(f(s'_1, q_1) = f(s'_2, q_2) = q' \wedge |s_1| \geq |s'_2| \wedge |s'_1| \leq |s_2| \right) \right) \quad (*) \end{aligned}$$

Aus der Gültigkeit dieser Beziehung folgt nämlich ohne weiteres die Gültigkeit der Beziehung in Definition 1.2.21. Der Beweis kann nun durch vollständige Induktion über den Längen von s_1 und s_2 erbracht werden.

- a) Sei $|s_1| = |s_2| = 1$. In diesem Fall gilt $(*)$, da laut Annahme P ein lokal konfluentes System ist.
- b) Annahme: Die Beziehung $(*)$ sei für alle s_1 und s_2 mit $|s_1| \leq K$ und $|s_2| \leq L$ ($K, L \in \mathbb{N}$) gültig.

Behauptung: Diese Beziehung ist dann auch für alle s_1 und s_2 mit $|s_1| \leq K+1$ und $|s_2| \leq L+1$ gültig.

Beweis:

Zum Beweis dieser Behauptung muß man zwischen drei Fällen unterscheiden:

Fall 1: $|s_1| = K+1$ und $|s_2| \leq L$

Seien $s_1 = p_{i_1} p_{i_2} \dots p_{i_{K+1}} = \tilde{s}_1 p_{i_{K+1}}$ und $s_2 = p_{j_1} p_{j_2} \dots p_{j_L}$

zwei beliebige Sequenzen und q ein beliebiger Zustand mit $f(s_1, q) = q_1$, $f(s_2, q) = q_2$ und $q_1 \neq q_2$. Ist

$f(\tilde{s}_1, q) = q_1$ und $q_1 = q_2$, so gilt trivialerweise Beziehung $(*)$ mit $s'_1 = \tilde{s}_1$ und $s'_2 = p_{i_{K+1}}$. Ist $q_1 \neq q_2$,

so gilt laut Induktionsannahme die Beziehung:

$$f(\tilde{s}_1, q) = q_1 \wedge f(s_2, q) = q_2 \wedge q_1 \neq q_2 \Rightarrow$$

$$\Rightarrow \exists q' \exists s'_1 \exists s'_2 (f(s'_1, q) = f(s'_2, q) = q' \wedge \tilde{s}_1 \neq s'_1 \wedge |s'_1| \leq |s_1| \wedge |s'_2| \leq |s_2|)$$

Schließt man den trivialen Fall aus, in dem $p_{i_{K+1}}$ das erste Element der Sequenz s'_1 ist, so ergibt

sich folgende Situation: Vom Zustand q_1 aus führt die Sequenz $P_{i_{K+1}}$ zum Zustand q'_1 und die Sequenz s'_1 zum Zustand q' . Für die Längen dieser Sequenzen gilt: $|P_{i_{K+1}}| = 1 \leq K$ und $|s'_1| \leq |s_2| = L$. Laut Induktionsannahme muß es dann einen Zustand q'' und zwei Sequenzen s''_1 und s''_2 mit $|s''_1| \leq L$ und $|s''_2| \leq 1$ geben, so daß gilt: $f(s''_1, q'_1) = f(s''_2, q') = q''$.

Somit ist $f(s'_2 s''_2, q_2) = f(s''_2, f(s'_2, q_2)) = f(s''_2, q') = q''$.

Die Sequenz $s'_2 s''_2$ mit $|s'_2 s''_2| \leq K+1$ führt also vom Zustand q_2 zum Zustand q'' .

Fall 2: $|s_1| = K$ und $|s_2| = L+1$

Der Beweis verläuft analog zu Fall 1.

Fall 3: $|s_1| = K+1$ und $|s_2| = L+1$

Seien $s_1 = P_{i_1} P_{i_2} \dots P_{i_{K+1}} = \tilde{s}_1 P_{i_{K+1}}$ und $s_2 = P_{j_1} P_{j_2} \dots P_{j_{L+1}} = \tilde{s}_2 P_{j_{L+1}}$ zwei beliebige Sequenzen und q ein beliebiger Zustand. Der triviale Fall $f(\tilde{s}_1, q) = f(\tilde{s}_2, q)$ wird wie im Fall 1) behandelt. Ist $f(\tilde{s}_1, q) \neq f(\tilde{s}_2, q)$, so gilt laut Induktionsannahme die Beziehung:

$$f(\tilde{s}_1, q) = \tilde{q}_1 \wedge f(\tilde{s}_2, q) = \tilde{q}_2 \wedge \tilde{q}_1 \neq \tilde{q}_2 \Rightarrow$$

$$\exists q' \exists \tilde{s}'_1 \exists \tilde{s}'_2 (f(\tilde{s}'_1, \tilde{q}_1) = f(\tilde{s}'_2, \tilde{q}_2) = q' \wedge |\tilde{s}'_1| \geq |\tilde{s}'_2| \wedge |\tilde{s}'_1| \leq |\tilde{s}'_2|)$$

Es sei weiterhin

$f(P_{i_{K+1}}, \tilde{q}_1) = q_1 \wedge f(P_{j_{L+1}}, \tilde{q}_2) = q_2$, wobei alle vier Zustände $q_1, \tilde{q}_1, q_2, \tilde{q}_2$ verschieden sind.

Vom Zustand $\tilde{q}_1$ aus führt die Sequenz $P_{i_{K+1}}$ mit der Länge

Eins zum Zustand q_1 und die Sequenz $\tilde{s}_1'$ mit $|s_1'| \leq L$

zum Zustand q' , wobei $q_1 \neq q'$ gilt.

Laut Induktionsannahme gilt also:

$$\exists q_1' \exists \tilde{s}_1'' \exists t_1 (f(\tilde{s}_1'', q_1) = f(t_1, q') = q_1' \wedge |\tilde{s}_1''| \leq L \wedge |t_1| \leq 1)$$

Vom Zustand $\tilde{q}_2$, andererseits, führt die Sequenz $P_{j_{L+1}}$

mit der Länge Eins zum Zustand q_2 und die Sequenz $\tilde{s}_2'$

mit $|s_2'| \leq K$ zum Zustand q' , wobei $q' \neq q_2$ gilt.

Laut Induktionsannahme gilt also:

$$\exists q_2' \exists \tilde{s}_2'' \exists t_2 (f(\tilde{s}_2'', q_2) = f(t_2, q') = q_2' \wedge |\tilde{s}_2''| \leq K \wedge |t_2| \leq 1).$$

Dies bedeutet, daß vom Zustand q' die Sequenz t_1 (bzw. t_2) zum Zustand q_1' (bzw. q_2') führt, wobei $|t_1| \leq 1$ und $|t_2| \leq 1$ gilt.

Laut Induktionsannahme gilt also:

$$\exists q_3 \exists t_1' \exists t_2' (f(t_1', q_1') = f(t_2', q_2') = q_3 \wedge |t_1'| \leq |t_2'| \leq 1)$$

Gilt also die Beziehung

$$f(s_1, q) = q_1 \wedge f(s_2, q) = q_2 \wedge |s_1| = K+1 \wedge |s_2| = L+1 \wedge q_1 \neq q_2,$$

dann gibt es einen Zustand q_3 und zwei Sequenzen

$\tilde{s}_1'' t_1'$ und $\tilde{s}_2'' t_2'$ mit $|\tilde{s}_1'' t_1'| \leq L+1$, $|\tilde{s}_2'' t_2'| \leq K+1$ und

$$f(\tilde{s}_1'' t_1', q_1) = f(\tilde{s}_2'' t_2', q_2) = q_3.$$

Es gilt nämlich:

$$f(\tilde{s}_1^* t_1^i, q_1) = f(t_1^i, f(\tilde{s}_1^*, q_1)) = f(t_1^i, q_1^i) = q_3 ,$$

$$f(\tilde{s}_2^* t_2^i, q_2) = f(t_2^i, f(\tilde{s}_2^*, q_2)) = f(t_2^i, q_2^i) = q_3 .$$

Es sei nochmals betont, daß in allen drei Fällen die Annahme gemacht wurde, daß die Ausführung von zwei verschiedenen Sequenzen von einem Zustand aus zu zwei verschiedenen Zuständen führt. Gilt dies nicht, so sind die Beweise trivial.

Damit ist Satz 1.2.22 vollständig bewiesen.

Jedes global konfluente Prozeßsystem ist auch lokal konfluent, da die lokale Konfluenz ein Spezialfall der globalen Konfluenz ist. Aus dem letzten Satz folgt somit die Gültigkeit der nächsten Aussage.

1.2.23 Korollar

Ein Prozeßsystem ist genau dann global konfluent, wenn es lokal konfluent ist.

1.2.24 Satz

Jedes halbkommutative und persistente Prozeßsystem ist konfluent.

Beweis

Sei P ein beliebiges halbkommutatives und persistentes Prozeßsystem. Weiterhin seien $P_i, P_j \in \bar{P}$ und $q, q_1, q_2 \in Q$, mit $i \neq j$ und $q_1 \neq q_2$.

Da das System persistent ist, folgt aus

$$f(P_i, q) = q_1 \neq q \text{ und } f(P_j, q) = q_2 \neq q$$

die Beziehung:

$$f(P_1, f(P_j, q)) \neq f(P_j, q) \quad \text{und}$$

$$f(P_j, f(P_1, q)) \neq f(P_1, q), \text{ d.h.}$$

$$f(P_1, q_2) \neq q_2 \quad \text{und} \quad f(P_j, q_1) \neq q_1.$$

Da aber das zugrundegelegte Prozeßsystem halbkommutativ ist, muß

$$f(P_1 P_j, q) = f(P_j P_1, q) \quad \text{gelten, d.h.}$$

$$\exists q' (f(P_1, q_2) = q' \text{ und } f(P_j, q_1) = q').$$

Das System ist also (lokal) konfluent.

2. Eine universale Basis für Betriebssysteme

2.1 Einleitung

In diesem Abschnitt wird eine mathematische Maschine zur Modellierung von Betriebssystemen entwickelt. Es werden in erster Linie Gesichtspunkte untersucht, die mit der Parallelität und Synchronisation zusammenhängen. Somit werden andere für die Betriebssysteme wichtige Aspekte wie z.B. Speicherverwaltung und Datenschutz außer Acht gelassen. Auch diese Bereiche könnten mit Hilfe der hier entwickelten Maschine behandelt werden. Da dies aber nicht explizit geschieht, wird von einer Betriebsmittel-Maschine und nicht von einer Betriebssystem-Maschine gesprochen.

Ausgangspunkt der Überlegungen in diesem Kapitel wird der im ersten Kapitel eingeführte Begriff "Prozeß" bzw. "Prozeßsystem" sein. Außerdem werden weitere Begriffe eingeführt und behandelt, deren Rechtfertigung in ihrer praktischen Zweckmäßigkeit liegt. Von einem mathematischen Modell eines physikalischen Prozesses erwartet man nämlich einen offensichtlichen Bezug des Modells zum physikalischen Prozeß.

Im folgenden wird die Frage der Entscheidbarkeit (Lösbarkeit) von fundamentalen Problemen aus der Theorie und Praxis der Betriebssysteme genauer untersucht. Die Bedeutung von Aussagen über die Lösbarkeit bzw. Entscheidbarkeit von gewissen Problemen innerhalb eines mathematischen Modells ist offensichtlich: Ist nämlich gezeigt worden, daß ein gewisses Problem nicht lösbar bzw. nicht partiell lösbar ist, so können damit vergebliche Lösungsversuche dieses Problems erspart bleiben. In diesem Falle bleibt dann nichts anderes

übrig, als das zugrundegelegte Modell soweit einzuschränken, bis das untersuchte Problem lösbar wird. Ist aber ein gewisses Problem im Rahmen eines vorgegebenen Modells lösbar, dann kann man sich um die Ermittlung einer Lösung bemühen, falls dies nicht bereits bei der Durchführung des Existenzbeweises geschehen ist.

2.2 Konstruktion einer Betriebsmittel-Maschine

2.2.1 Definition

Sei $D = |N_0^n$ mit $n \in |N$. Eine Funktion f von D in D heißt Elementarfunktion erster Art, wenn es einen festen Vektor $(c_1, c_2, \dots, c_n) \in |N_0^n$ gibt, so daß für alle $d = (d_1, d_2, \dots, d_n) \in D$ gilt:

$$f(d) = (d_1 + c_1, d_2 + c_2, \dots, d_n + c_n).$$

Die partielle Funktion f heißt Elementarfunktion zweiter Art, wenn es einen festen Vektor $(c_1, c_2, \dots, c_n) \in |N_0^n$ gibt, so daß für alle $d = (d_1, d_2, \dots, d_n) \in D$ gilt

$$f(d) = \begin{cases} (d_1 - c_1, d_2 - c_2, \dots, d_n - c_n) & \forall i (1 \leq i \leq n \Rightarrow d_i \geq c_i) \\ \text{falls} & \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Auch in diesem Abschnitt werden nur funktionale Prozesse betrachtet.

2.2.2 Definition

Ein Prozeß (siehe Def. 1.1.1) mit der Adressenmenge $A = \{a_0, a_1, \dots, a_n\}$ heißt Elementarprozeß, wenn gilt:

- a) Für alle nach Def. 1.1.6 eingeführten Operationen O_{ij} mit $i, j \in IA$ ($IA = \{0, \dots, n\}$) gilt:

$$|O_{ij}| \neq 0 \iff j = (i+1) \bmod (n+1).$$

Für alle nichtleeren Operationen O_{ij} ist j also eindeutig aus i bestimmt. Man kann somit O_i statt O_{ij} schreiben.

b) O_0 ist eine Elementarfunktion zweiter Art.

c) O_i ist eine Elementarfunktion erster Art ($1 \leq i \leq n$).

Bemerkungen

Das Programm eines Elementarprozesses ist zyklisch. Deswegen braucht man die Adressen nicht explizit hinzuschreiben. Außerdem werden bei Ausführung einer Operation nur die Werte der Variablen d_i geändert, für die $c_i \neq 0$ gilt (Definition 2.2.1). Werden bei einer Ausführung einer Operation erster Art nur die Werte der Variablen $d_{i_1}, \dots, d_{i_K}$ verändert, so kann man für diese Operation folgende Notation gebrauchen:

$$(d_{i_1}, \dots, d_{i_K}) := (d_{i_1} + c_{i_1}, \dots, d_{i_K} + c_{i_K}).$$

Gilt weiterhin $c_{i_1} = c_{i_2} = \dots = c_{i_K} = 1$, so kann man für diese Notation auch

$$f_+(d_{i_1}, d_{i_2}, \dots, d_{i_K})$$

schreiben.

Werden bei der Ausführung einer Elementaroperation zweiter Art nur die Werte der Variablen $d_{i_1}, d_{i_2}, \dots, d_{i_K}$ verändert, so wird für diese Operation folgende Notation eingeführt:

only if $d_{i_1} \geq c_{i_1} \wedge d_{i_2} \geq c_{i_2} \wedge \dots \wedge d_{i_K} \geq c_{i_K}$ do

$$\{d_{i_1} := d_{i_1} - c_{i_1}; \dots; d_{i_K} := d_{i_K} - c_{i_K}\}$$

Ist die Bedingung in dieser Operation nicht erfüllt und versucht ein Prozeß diese Operation auszuführen, so bleibt er an dieser Stelle blockiert. Ist weiterhin in dieser Operation $c_{i_1} = c_{i_2} = \dots = c_{i_K} = 1$, so kann man die obere Notation durch

$$f_{-}(d_{i_1}, d_{i_2}, \dots, d_{i_K})$$

vereinfachen.

Sei nun $\bar{P} = \{P_1, P_2, \dots, P_K\}$ eine vorgegebene Menge von Elementarprozessen mit

$$P_i = (A_i, \underline{D}_i, R_i, a_{iO}, \underline{D}_{iO}) \quad (1 \leq i \leq K)$$

und $P = (\underline{A}, \underline{D}, R, \underline{a}_O, \underline{D}_O)$

ein nach vorigem Abschnitt aus $\bar{P}$ konstruiertes Prozeßsystem.

2.2.3 Definition

Ist $\underline{D} = \tilde{D}_1 \times \tilde{D}_2 \times \dots \times \tilde{D}_M$ die Datenmenge von P und $\underline{v} = (\tilde{v}_1, \tilde{v}_2, \dots, \tilde{v}_M)$ die Datenvariable von P , so heißt die Menge $BM = \{b_1, b_2, \dots, b_L\}$ die Menge der Betriebsmittel von P und b_i ($1 \leq i \leq L$) das i -te Betriebsmittel, wenn BM eine Zerlegung der Menge $\{\tilde{v}_1, \tilde{v}_2, \dots, \tilde{v}_M\}$ darstellt.

Ein Betriebsmittel enthält also eine nichtleere Menge von Variablen des Prozeßsystems, wobei jede dieser Variablen in genau einem Betriebsmittel vorkommt.

2.2.4 Definition

Eine Elementarfunktion f von $\underline{D}$ in $\underline{D}$ greift genau dann auf die Datenvariable $\tilde{v}_i$ ($1 \leq i \leq M$) von P zu, wenn für die Komponente c_i des festen Vektors $(c_1, c_2, \dots, c_M)$ - siehe Definition 2.2.1 - von f gilt $c_i \geq 1$

2.2.5 Definition

Eine Elementarfunktion heißt "erlaubt bezüglich der Menge BM ", wenn sie nur auf solche Variablen zugreift, die in einem Betriebsmittel $b_i \in BM$ enthalten sind.

Im folgenden werden nur erlaubte Elementarfunktionen betrachtet.

2.2.6 Definition

Sei $b_i \in BM$ ein Betriebsmittel. Die Menge ZB_i enthalte alle Elementarprozesse, die durch eine erlaubte Elementar-Operation zweiter Art auf b_i zugreifen. Eine transitive, irreflexive und nichtsymmetrische Relation PR_i , mit $PR_i \subseteq ZB_i \times ZB_i$ heißt dann Prioritätsrelation bez. b_i .

Man sagt auch, daß P_K bezüglich b_i eine höhere Priorität besitzt als P_l (in Zeichen $P_K \succ P_l$), wenn gilt:

$$(P_K, P_l) \in PR_i$$

Unter der Irreflexivität der Prioritätsrelationen versteht man folgendes: für alle i ($1 \leq i \leq L$) und alle l ($1 \leq l \leq K$) gilt:

$$(P_l, P_l) \notin PR_i$$

Die Nichtsymmetrie der Prioritätsrelationen bedeutet folgendes:

Aus $(P_K, P_1) \in PR_1$ folgt $(P_1, P_K) \notin PR_1$.

Besteht die Menge ZB_1 aus einem einzigen Element, so ist PR_1 leer. Die Relation PR_1 kann aber auch dann leer sein, wenn ZB_1 aus mehreren Elementen besteht. In diesem Fall sagt man, daß unter den auf b_1 zugreifenden Prozessen keine Prioritätsverhältnisse herrschen.

2.2.7 Definition

Sei P ein aus den Elementar-Prozessen $\{P_1, P_2, \dots, P_K\}$ nach Definition 1.2.10 zusammengesetztes Prozeßsystem. Weiterhin sei $BM = \{b_1, b_2, \dots, b_L\}$ eine Menge von Betriebsmitteln nach Definition 2.2.3 und $PR = \{PR_1, PR_2, \dots, PR_L\}$ eine nach Definition 2.2.6 Menge von Prioritätsrelationen. Das Tripel $\mathcal{L} = (P, BM, PR)$ nennt man eine Betriebsmittel-Maschine.

Im folgenden wird gezeigt, daß diese mathematische Maschine eine universale Basis für Betriebssysteme bildet. Zunächst aber folgt die Beschreibung der Arbeitsweise (Semantik) dieser Maschine. Dazu wird auf die Ausführungen des ersten Kapitel zurückgegriffen, wobei aber zusätzliche Angaben erforderlich sind. In $\mathcal{L}$ sind nämlich die neuen Größen BM und PR enthalten; außerdem besitzen die Elementarprozesse eine standardisierte Form.

Zur Vereinfachung der Darstellung der Arbeitsweise der Maschine $\mathcal{L}$ werden zunächst einige Vereinbarungen getroffen. Liegt ein Zustand $q \in Q$ des Prozeßsystems vor, so sagt man, Prozeß $P_K \in \bar{P}$ befindet sich an seiner i -ten Stelle bzw. i -ten Adresse, wenn (siehe Def. 1.2.2)

$$\pi_{\alpha}(q, k) = a_{ki}$$

Dieser Stelle des betrachteten Prozesses entspricht eindeutig eine erlaubte Elementar-Funktion erster Art (zweiter Art), falls $i \neq 0$ (bzw. $i = 0$) ist. Der Einfachheit wegen werden hier statt der Datenvariablen eines Prozesses die durch Id bestimmten Datenvariablen des Prozeßsystems betrachtet (siehe auch Bemerkungen im Anschluß an die Definition 1.2.3). Im folgenden wird der Begriff "lauffähig" gegenüber der Definition 1.2.12 leicht modifiziert und der Zustand "wartend" neu eingeführt. Ein Prozeß, der bezüglich eines momentanen Zustandes q nicht lauffähig ist, heißt blockiert. Ein lauffähiger Prozeß, der die Ausführung einer Operation begonnen, aber noch nicht beendet hat, heißt laufend.

Ein lauffähiger Prozeß, der nicht laufend ist, heißt lauffähig (wartend), falls er mit allen (nicht mit allen) gerade laufenden Prozessen q -verträglich ist.

Zunächst wird angenommen, daß $|BM| = 1$ ist. Später wird diese Einschränkung aufgehoben. Das folgende Bild soll nun die hier eingeführten Bezeichnungen veranschaulichen und die Darstellung der Übergänge (siehe nächste Definition) erleichtern.

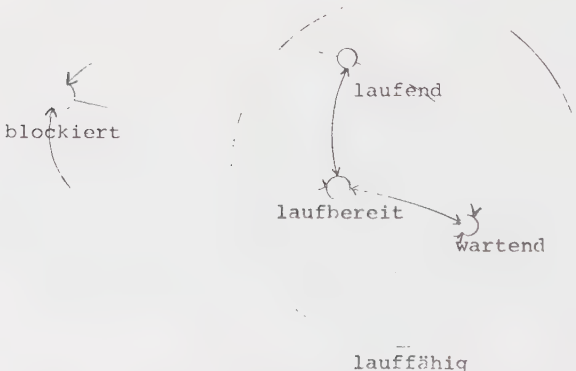


Bild 2.2.8: Zustandsdiagramm

Die nächste Definition gibt eine Beschreibung der Arbeitsweise von $\mathcal{L}$. Dabei tritt die Synchronisation von Prozessen bei ihrem Zugriff auf ein Betriebsmittel besonders in den Vordergrund, wobei unter Synchronisation die Verwaltung der Übergänge (gerichtete Pfeile) im Bild 2.2.8 zu verstehen ist. Der Einfachheit wegen wird $q(t)$ statt $\underline{y}(t)$ geschrieben.

2.2.9 Definition

Zu einem virtuellen Zeitpunkt t sei der Zustand des Prozeßsystems gleich $q(t)$, die Menge der laufenden Prozesse gleich $lp(t)$, die Menge der laufbereiten Prozesse $lfp(t)$ und die Menge der wartenden Prozesse gleich $wp(t)$.

a) für $t=0$ (unmittelbar nach der Initialisierung des Systems) gilt:

- 1) $lp(0)=\emptyset$, 2) $wp(0)=\emptyset$,
- 3) die Menge der blockierten Prozesse ist gleich $\bar{P} - lfp(0)$.

b) Die Mengen $lp(t)$, $lfp(t)$, $wp(t)$ seien zum Zeitpunkt t disjunkt. Die Menge $lp(t)$ sei $q(t)$ -verträglich. Weiterhin sei auch jeder Prozeß aus $lfp(t)$ mit $lp(t)$ $q(t)$ -verträglich. Für jeden Prozeß $P_i \in wp(t)$ gelte: es gibt einen Prozeß $P_j \in lp(t)$, so daß P_i und P_j nicht $q(t)$ -verträglich sind. Jeder Prozeß, der nicht in einer der Mengen $lp(t)$, $lfp(t)$ und $wp(t)$ enthalten ist, befinde sich im Zustand "blockiert". Der erweiterte momentane Zustand von $\mathcal{L}$ werde durch das Quadrupel $(q(t), lp(t), lfp(t), wp(t))$ bezeichnet. Findet zum Zeitpunkt $t+1$ eine Änderung dieses Zustandes statt,

$$(q(t), lp(t), lfp(t), w(t)) \rightarrow (q(t+1), lp(t+1), lfp(t+1), w(t+1)),$$

so kann man zwischen zwei Fällen unterscheiden:

β_1) Eine Menge $lfp' = \{P_{i_1}, P_{i_2}, \dots, P_{i_K}\} \subseteq lfp(t)$ wird zum Zeitpunkt $t+1$ in den Zustand "laufend" versetzt. Dieser Vorgang wird als "Starten" der Prozesse $P_{i_1}, P_{i_2}, \dots, P_{i_K}$ bezeichnet und kann nur dann stattfinden, wenn a) die Mengen $lp(t)$ und lfp' $q(t)$ -verträglich sind und b) es keinen Prozeß aus $lfp(t) - lfp'$ gibt, der eine höhere Priorität als irgendein Prozeß aus lfp' besitzt. Es gilt:

- 1) $q(t+1) = q(t)$
- 2) $lp(t+1) = lp(t) + lfp'$
- 3) $lfp(t+1) \subseteq lfp(t) - lfp'$
- 4) $wp(t+1) \supseteq wp(t)$.

Zur Beziehung 1) und 2) sind keine Erläuterungen nötig. Die Beziehung 3) hängt mit der Tatsache zusammen, daß die Verträglichkeitsrelation nicht transitiv ist. So ist z.B. möglich, daß die Menge $\{P_1, P_2\} \subseteq lfp(t)$ mit der Menge $lp(t)$ nicht $q(t)$ -verträglich ist. Wird also ein Prozeß $P_1 \in lfp(t)$ zum Zeitpunkt $t+1$ in den Zustand "laufend" versetzt ($P_1 \in lp(t+1)$), so werden alle Prozesse aus dem Zustand "laufbereit" entfernt und in den Zustand "wartend" versetzt, die mit dem Prozeß P_1 nicht q -verträglich sind. Somit ist jeder Prozeß aus $lfp(t+1)$ mit $lp(t+1)$ $q(t+1)$ -verträglich. Jeder Prozeß, der sich zum Zeitpunkt t im Zustand "war-

tend" befindet, verbleibt auch zum Zeitpunkt $t+1$ im selben Zustand (Beziehung 4).

Es sei zu betonen, daß lfp' nicht unbedingt alle Prozesse enthält, die zum Zeitpunkt $t+1$ hätten starten können. Dies soll den asynchronen Ablauf der einzelnen Prozesse der Maschine berücksichtigen.

β_2) Alle Prozesse einer Menge $\tilde{lp} = \{P_{i_1}, P_{i_2}, \dots, P_{i_K}\}$

$\subseteq lp(t)$ beenden zum Zeitpunkt $t+1$ ihre Aktionen.

Dieser Vorgang wird als "Beenden" der Prozesse

$\{P_{i_1}, P_{i_2}, \dots, P_{i_K}\}$ bezeichnet. Es gilt:

1) Aus $q(t)$ und $\tilde{lp}$ ist $q(t+1)$ eindeutig bestimmt.

Alle Elementarprozesse sind nämlich deterministisch und funktional, während andererseits die Menge $\tilde{lp}$ laut Konstruktion $q(t)$ -verträglich ist.

2) $lp(t+1) = lp(t) - \tilde{lp}$

3) $lfp(t) \subseteq lfp(t+1)$

4) Für einen Prozeß $P_i \in wp(t)$ gibt es nun drei Möglichkeiten:

4a) Prozeß P_i ist zwar bezüglich $q(t+1)$ lauffähig, aber mit $lp(t+1)$ nicht $q(t+1)$ -verträglich. In diesem Falle bleibt P_i auch zum Zeitpunkt $t+1$ im Zustand "wartend".

4b) P_i wird zum Zeitpunkt $t+1$ lauffähig. Die Menge aller solcher Prozesse sei mit wp' ($wp' \subseteq wp(t)$) bezeichnet. Dann ist $wp' \subseteq lfp(t+1)$.

4c) Prozeß P_i ist bezüglich $q(t+1)$ nicht mehr lauffähig. Er wird also zum Zeitpunkt $t+1$ in den Zustand "blockiert" versetzt. Die Menge aller solcher Prozesse sei mit wp'' ($wp'' \subseteq wp(t)$) bezeichnet.

- 5) Entsprechende Aussagen gelten für die Menge der zum Zeitpunkt t blockierten Prozesse, d.h.
- 5a) Manche Prozesse bleiben weiterhin auch zum Zeitpunkt $t+1$ blockiert.
- 5b) Eine Teilmenge bp' wird in den Zustand "laufbereit" versetzt, d.h. $bp' \subseteq lfp(t+1)$.
- 5c) Eine Teilmenge bp'' wird in den Zustand "wartend" versetzt, d.h. $bp'' \subseteq wp(t+1)$.
- 6) Ein Prozeß aus $\tilde{lp}$ kann zum Zeitpunkt $t+1$ blockiert, wartend oder laufbereit sein. Werden mit lp', lp'', lp''' die Mengen von Prozessen aus $\tilde{lp}$ bezeichnet, die zum Zeitpunkt $t+1$ wartend (lp'), laufbereit (lp'') oder blockiert (lp''') sind, so gilt: $\tilde{lp} = lp' + lp'' + lp'''$.

Beenden somit die Prozesse $P_{i_1}, P_{i_2}, \dots, P_{i_K}$ ihre

Aktionen zum Zeitpunkt $t+1$ (Vorgang β_2) so gilt:

$$lp(t+1) = lp(t) - \tilde{lp},$$

$$lfp(t+1) = lfp(t) + wp' + bp' + lp'' \quad \text{und}$$

$$wp(t+1) = wp(t) - wp' - wp'' + bp'' + lp'.$$

Zu jedem Zeitpunkt t ist die Menge der blockierten Prozesse gleich $\bar{P} - lp - lfp - wp$, so daß eine explizite Angabe dieser Menge nicht erforderlich ist.

Die Maschine $\mathcal{S}$ kann also zwei Arten von Aktionen ausführen, nämlich das "Starten" und "Beenden" von Prozessen. Diese halbformale Beschreibung wurde hier ausgewählt, um die anschauliche Vorstellung der Arbeitsweise von $\mathcal{S}$ zu vereinfachen.

Besteht die Menge BM von $\mathcal{L}$ aus mehreren Elementen, so ergibt dies keine Schwierigkeiten. Da die hier betrachteten Prozesse erlaubte Elementarprozesse sind, greift ein solcher Prozeß beim Vorliegen eines beliebigen Zustandes q auf die Variablen eines einzigen Betriebsmittels zu und ist völlig unabhängig (im Sinne von "Start- und Beende"-Aktionen) von anderen Prozessen, die auf ein anderes Betriebsmittel zugreifen. Zu jedem Zeitpunkt t kann man die Prozesse aus der Menge $\bar{P}$ in L Klassen $K_1, K_2, \dots, K_L$ (L = Anzahl der Betriebsmittel) so einteilen, daß ein Prozeß zu diesem Zeitpunkt genau dann der Klasse K_i angehört, wenn er im Zustand $q(t)$ auf das Betriebsmittel b_i zugreift. Für jede dieser Klassen erstellt man nun das Zustandsdiagramm von Bild 2.2.8. Ein Prozeß kann seine Klasse nur dann verlassen, wenn er im Zustand "laufend" gewesen ist und seine Aktion beendet hat. Tut er dies, dann wird er in die entsprechende Klasse aufgenommen und zwar in einem der Zustände "blockiert", "laufbereit" oder "wartend". Zu jedem virtuellen Zeitpunkt t gibt es also für jedes Betriebsmittel ein Zustandsdiagramm nach Bild 2.2.8. Solange ein Prozeß in seiner Klasse bleibt, können die einzelnen Änderungen innerhalb eines Zustandsdiagramms völlig unabhängig von anderen Zustandsdiagrammen stattfinden. Offensichtlich muß man eine Einschränkung machen: Die dem Betriebsmittel b_i entsprechende Relation PR_i muß nun auf die Menge K_i beschränkt werden. Dies bedeutet, daß bei den Startbedingungen (bezüglich q und b_i) nicht die Prioritätsrelation PR_i sondern PR'_i , wobei

$$PR'_i = PR_i \cap K_i \times K_i \text{ ist,}$$

herangezogen werden muß. Dadurch wurde erreicht, daß

die entscheidende Prioritätsrelation PR'_1 eine dynamische Struktur erhält, da PR'_1 mit Rücksicht auf den momentanen Zustand q erstellt wird.

2.2.10 Bemerkungen zu der Arbeitsweise von $\mathcal{L}$

Aus den bisherigen Ausführungen ist es offensichtlich, daß die hier entwickelte Betriebsmittel-Maschine ein adäquates Modell für Betriebssysteme ist. Das Modell gestattet insbesondere die parallele Arbeit von verschiedenen Teilen (Prozessen) des Systems.

Die Änderungen eines momentanen Zustandes q werden von den einzelnen Prozessen der Maschine ausgeführt. Während der Arbeit von $\mathcal{L}$ werden aber auch Änderungen der Prozeßzustände laut Bild 2.2.8 durchgeführt, die hier nur funktional beschrieben wurden. Bei der Implementierung dieses Modells muß somit für jedes Betriebsmittel b_i eine Instanz M_i (Monitor für das Betriebsmittel b_i) existieren, die die Änderungen der Prozeßzustände innerhalb der Klasse K_i durchführt. Ein Monitor darf aber keine Änderung an q ausführen. Insbesondere hat er dafür zu sorgen, daß beim Übergang von Prozeßzustand "laufbereit" zum Zustand "laufend" die Prioritätsrelation PR'_1 nicht verletzt wird. Aus der Konstruktion und Arbeitsweise von $\mathcal{L}$ ist folgender Sachverhalt offensichtlich: Die Menge K_i enthält zu einem virtuellen Zeitpunkt t alle Prozesse, die auf b_i zugreifen. Prioritätsverhältnisse können aber nur unter denjenigen Prozessen aus K_i bestehen, die mit einer erlaubten Operation zweiter Art auf b_i zugreifen. Alle

anderen Prozesse aus K_1 können vom Monitor beliebig und unabhängig von Einschränkungen durch die Prioritätsrelation gewählt werden. Elementaroperationen erster Art modellieren unter anderem die Freigabe von bereits belegten Betriebsmitteln und die Erzeugung von Nachrichten. Die Elementaroperationen zweiter Art sollen unter anderem die Belegung von Betriebsmitteln und den Verbrauch von Nachrichten modellieren. Die Maschine $\mathcal{L}$ wurde so konstruiert, daß dem Programmierer gewisse Freiheiten bei der Konstruktion der Monitore übrig bleiben, obwohl $\mathcal{L}$ wie im folgenden gezeigt wird, eine universale Basis für Betriebssysteme ist. Dies bedeutet folgendes: Gibt es zu einem Problem ein Betriebssystem, welches eine Lösung dieses Problems darstellt, so gibt es auch eine Betriebsmittel-Maschine, die dieses Problem löst.

An dieser Stelle sei kurz auf eine strukturelle Erweiterung von $\mathcal{L}$ hingewiesen, die aber keine Erhöhung der Leistungsfähigkeit von $\mathcal{L}$ bewirkt. Besitzt $\mathcal{L}$ L Betriebsmittel und somit L Prioritätsrelationen bzw. L Monitore, so kann man eine weitere Instanz, Supervisor genannt, einführen. Ähnlich zum Monitor, der einzelne Prozesse überwacht, trifft der Supervisor Entscheidungen über die Reihenfolge, in der die Monitore ihre Auswahl treffen sollen. Dies kann der Supervisor aufgrund einer Prioritätsrelation tun, die eine Teilmenge von

$$\{(M_i, M_j) \mid i \neq j \wedge 1 \leq i, j \leq L\}$$

ist. Dies würde einen hierarchischen Aufbau der Prozesse und der obengenannten Instanzen mit sich bringen, wie folgendes Bild veranschaulichen soll:

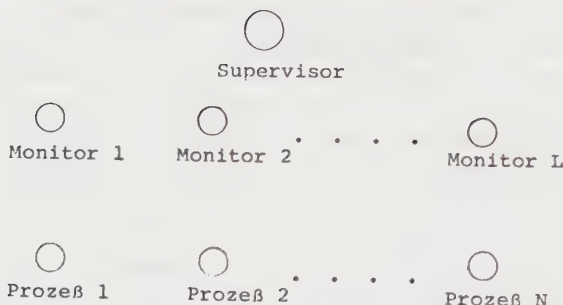


Bild 2.2.11: Hierarchischer Aufbau von $\mathcal{L}$

Da eine völlige Analogie zwischen Supervisor und Monitor besteht, wird hier auf eine ausführliche Behandlung des Supervisors bzw. der entsprechenden Relation verzichtet.

Als nächstes wird Korollar 2.2.26 bewiesen, nach dem die Maschine $\mathcal{L}$ eine universale Basis für Betriebssysteme bildet. Dazu wird zuerst gezeigt, daß jede Turing-Maschine durch eine Betriebsmittel-Maschine simuliert werden kann. Der Beweis läßt sich aber einfacher gestalten, wenn man statt der Turing-Maschine die Programm-Maschine mit zwei Registern zugrunde legt.

In Satz 2.2.13 wird zu einer beliebigen Programm-Maschine Π mit zwei Registern eine Betriebsmittel-Maschine konstruiert, die "alles machen kann, was Π macht". Dabei wird Maschine $\mathcal{L}$ umfangreicher sein als Π . Dies ist dadurch zu erklären, daß $\mathcal{L}$ im allgemeinen für parallele Abläufe konstruiert wurde, während Π nur sequentiell arbeitet.

An dieser Stelle wird nur eine kurze Beschreibung der Programm-Maschine angegeben. Diese Maschine wird in [57] genauer untersucht.

Eine Programm-Maschine Π besitzt zwei Register r_1 und r_2 , deren Inhalte beliebige Zahlen aus $\mathbb{N}_0$ sein können. Der Einfachheit wegen werden dieselben Bezeichnungen für Register und Registerinhalte benutzt. Zur Maschine gehört auch ein Programm mit einer endlichen Anzahl von adressierten Befehlen. Für dieses Programm sind nur folgende Befehlstypen zugelassen:

- 1) r^+ : Erhöhe den Inhalt des Registers r ($r \in \{r_1, r_2\}$) um eins und gehe zum nächsten Befehl weiter.
- 2) $r^-(a_n)$: Ist $r \geq 1$, so ersetze r durch $r-1$ und gehe zum nächsten Befehl weiter. Ist $r=0$, so gehe zum n -ten Befehl weiter.
- 3) HALTE: Maschine anhalten.

Offensichtlich bestehen gewisse Ähnlichkeiten zwischen den Elementaroperationen von $\mathcal{L}$ und den zwei ersten Befehlstypen von Π . Ein erster Unterschied besteht darin, daß für die Elementar-Prozesse von $\mathcal{L}$ als nächster Befehl immer Addition modulo n genommen wird. Kommt Π während einer Berechnung an eine Stelle des Programms, wo ein Befehl vom Typ 2 ist, so ist auch sicher, daß dieser Befehl ausgeführt werden kann. Dieser Befehlstyp realisiert also eine totale Funktion über der Zustandsmenge von Π . Die Elementar-Operationen zweiter Art von $\mathcal{L}$ sind aber im allgemeinen partielle Funktionen. Im folgenden werden ausschließlich Programm-Maschinen mit zwei Registern betrachtet. Wird ein zyklischer Prozeß von seiner Anfangsadresse aus gestartet und kehrt danach wieder an seine Anfangsadresse zurück, so sagt man, daß dieser Prozeß einen vollen Umlauf (Prozeßumlauf) ausgeführt hat.

2.2.12 Definition

Sei Π eine Programm-Maschine mit zwei Registern r_1 und r_2 . Weiterhin sei $\mathcal{L}$ eine Betriebsmittel-Maschine mit zwei ausgezeichneten Variablen x_1 und x_2 , wobei

$\text{Dom}(x_1) = \text{Dom}(x_2) = \mathbb{N}_0$ ist. $\mathcal{L}$ simuliert Π , wenn gilt:

- 1) Wird die Variable x_i mit demselben Wert initialisiert wie Register r_i ($i \in \{1, 2\}$) und führt Π k Befehle aus ($k \geq 1$), so wird auch $\mathcal{L}$ k Prozeßumläufe ausführen, so daß gilt: $x_i = r_i$ für $i = 1, 2$.
- 2) Hält Π nach k Schritten an, dann wird $\mathcal{L}$ nach $k+1$ Schritten in einen Zustand geraten, in dem sämtliche Prozesse blockiert sind.

2.2.13 Satz

Zu jeder beliebig vorgegebenen Programm-Maschine Π kann man eine Betriebsmittel-Maschine $\mathcal{L}$ konstruieren, so daß Maschine $\mathcal{L}$ die Maschine Π simuliert.

Beweis

Der Beweis wird in mehreren Schritten durchgeführt und endet mit Lemma 2.2.21.

Sei nun Π eine beliebig vorgegebene Programm-Maschine, die aus n adressierten Befehlen besteht. Es wird o.B.d.A. angenommen, daß der HALTE-Befehl als n -ter Befehl des Programms von Π auftritt. Man kann nun eine Betriebsmittel-Maschine $\mathcal{L}$ konstruieren, die Π simuliert.

Konstruktion der Datenbasis von $\mathcal{L}$:

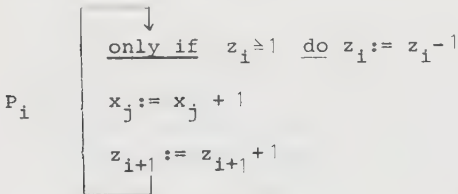
Man führt die Variablen $x_1, x_2, z_1, z_2, \dots, z_n$ mit

$\text{Dom}(x_1) = \text{Dom}(x_2) = \mathbb{N}_0$ und $\text{Dom}(z_i) = \{0, 1\}$ für $1 \leq i \leq n$.

Konstruktion der Elementar-Prozesse von $\mathcal{L}$

Hier wird eine Vorschrift zur Konstruktion einer endlichen Anzahl von Elementarprozessen für $\mathcal{L}$ gegeben. Es wäre möglich, mit einer kleineren Anzahl von solchen Prozessen auszukommen. Da es sich in diesem Beweis aber um eine Existenzaussage handelt, spielt die Anzahl der Prozesse von $\mathcal{L}$ keine Rolle. Dem i -ten Befehl von Π ($1 \leq i \leq n$) ordne man einen bzw. zwei Elementarprozesse nach folgender Vorschrift zu.

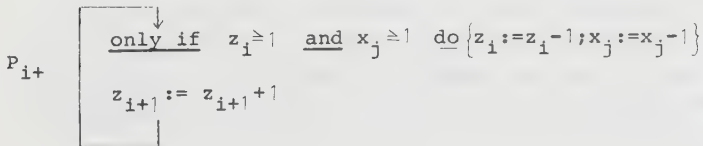
- 1) Der i -te Befehl von Π sei von der Form $i \ x_j^+$, mit $j \in \{1, 2\}$.

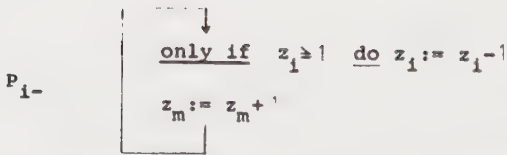


Befindet sich dieser Prozeß an seiner ersten Adresse und ist die dort aufgeführte Bedingung nicht erfüllt, so bleibt er weiterhin an dieser Stelle stehen.

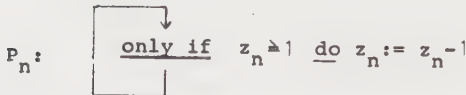
- 2) Der i -te Befehl von Π sei von der Form $i \ x_j^-(a_m)$, mit $j \in \{1, 2\}$.

Man konstruiere jetzt zwei Elementarprozesse P_{i+} und P_{i-} der Form:





- 3) Dem HALTE-Befehl von Π , der laut Vereinbarung der n -te Befehl von Π ist, ordne man folgenden Elementarprozeß zu:



Konstruktion der Betriebsmittel und der Prioritätsrelationen von $\mathcal{L}$

Da die Programm-Maschine Π sequentiell arbeitet, ist es sinnvoll, sämtliche Variablen von $\mathcal{L}$ in einem einzigen Betriebsmittel zusammenzufassen und somit eine Prioritätsrelation aufzustellen. Die Prioritätsverhältnisse werden folgendermaßen festgelegt: Besitzt der i -te Befehl von Π die Form $i \ r_j^- (a_m)$, so besitzt Prozeß P_{i+} eine höhere Priorität als P_{i-} . Dies heißt, befinden sich zu einem beliebigen Zeitpunkt beide Prozesse im Zustand "laufbereit", so wird nur Prozeß P_{i+} in den Zustand "laufend" versetzt. Weitere Prioritätsverhältnisse bestehen nicht.

Initialisierung der Datenbasis von $\mathcal{L}$

Die Variable x_i von $\mathcal{L}$ wird mit demselben Wert initialisiert wie Register r_i ($i \in \{1, 2\}$) von Π . Man kann o.B.d.A annehmen, daß das Programm der Maschine Π immer mit dem Befehl der Adresse a_1 startet. Die Initialisierung der übrigen Variablen sieht dann folgendermaßen aus:

$$z_1 := 1, z_i := 0 \text{ für } 2 \leq i \leq n.$$

Nach der Initialisierung kann Maschine $\mathcal{L}$ anfangen zu laufen. Aus der Konstruktion von $\mathcal{L}$ sind folgende Aussagen ersichtlich.

2.2.14 Lemma

Zu jedem Zeitpunkt kann höchstens ein Elementar-Prozeß im Zustand "laufend" sein.

Beweis

Der Beweis ist sehr einfach und kann durch vollständige Induktion durchgeführt werden. Laut Initialisierung der Datenbasis von $\mathcal{L}$ gilt diese Aussage zum virtuellen Zeitpunkt $t=0$ (Startzeit von $\mathcal{L}$). Befindet sich zu einem virtuellen Zeitpunkt t ein Prozeß im Zustand "laufend", so kann nach seiner Beendigung höchstens ein Prozeß in den Zustand laufend versetzt werden.

Befindet sich ein Prozeß an seiner Anfangsadresse und wird in den Zustand "laufend" versetzt, so ist es auch sicher, daß er wieder an seine Anfangsadresse gelangen kann. Andererseits ist auch folgende Aussage so einleuchtend, daß sich deren Beweis erübrigt.

2.2.15 Lemma

Befindet sich ein Elementar-Prozeß an seiner Anfangsadresse und wird in den Zustand "laufend" versetzt, dann kann er genau einen Umlauf ausführen. Erst wenn ein anderer Prozeß einen Umlauf ausgeführt hat, kann der erste wiederum gestartet werden. Hier wurde o.B.d.A. von der Annahme ausgegangen, daß für alle Befehle von Typ 2 der Maschine gilt:

$$a_m : r^{-1}(a_n) \Rightarrow m \neq n.$$

Die beiden Lemmata gestatten folgende Vereinfachung:
Der Adreßteil eines momentan vorliegenden Zustandes q von $\mathcal{L}$ kann ohne weiteres vernachlässigt werden. Entscheidend ist nur der Datenteil von q . Befindet sich nämlich ein Prozeß an seiner ersten Adresse und wird er gestartet, so ist es gesichert, daß er einen vollen Umlauf ausführen kann, ohne daß zwischendurch ein anderer Prozeß gestartet wird. Für die Zustandsmenge Q kann man also schreiben:

$$Q = \{(x_1, x_2, z_1, z_2, \dots, z_n) \mid x_1, x_2 \in \mathbb{N}_0, z_i \in \{0, 1\} \text{ für } 1 \leq i \leq n\}$$

2.2.16 Definition

Sei $q \in Q$ und alle Prozesse der hier entwickelten Maschine $\mathcal{L}$ befänden sich jeweils an ihren ersten Adressen. Kann im Zustand q Prozeß P_i in den Zustand "laufend" versetzt werden, so ist unter $p_i(q)$ der Zustand zu verstehen, der sich aus q nach einem vollen Umlauf von P_i ergibt.

Als nächstes wird der Begriff der ausführbaren Sequenz $p_{i_1} p_{i_2} \dots p_{i_l}$ eingeführt, wobei der Einfachheit wegen statt p_{i_j} einfach p_j geschrieben wird.

2.2.17 Definition

Sei $q_0 \in Q$. Dann ist die leere Sequenz Λ ausführbar und gelte $\Lambda(q_0) = q_0$. Ist die Sequenz $x = p_1 p_2 \dots p_k$ von q_0 aus ausführbar und gilt $x(q_0) = p_k(\dots p_2(p_1(q_0))\dots) = q$, dann ist die Sequenz $x' = x p_{k+1}$ genau dann von q_0 aus ausführbar, wenn Prozeß P_{k+1} von q aus in den Zustand "laufend" versetzt werden kann.

Dann gilt:

$$x'(q_0) = xp_{k+1}(q_0) = p_{k+1}(q).$$

Lauf Definition kann die Programm-Maschine Π eine eindeutige (endliche oder unendliche) Folge von Befehlen ausführen, die vom Programm von Π und von der Initialisierung der Register abhängt. Diese Folge wird eine ausführbare Sequenz von Befehlen von Π genannt und mit $y = t_1 t_2, \dots, t_i, \dots$ bezeichnet, wobei hier wiederum statt t_{ij} einfach t_j geschrieben wird. Startet die Programm-Maschine immer mit dem ersten Befehl ihres Programms, so ist in jeder ausführbaren Sequenz von Π t_1 der erste Befehl, während t_i ($i \neq 1$) irgendein Befehl des Programms von Π ist. Werden die Register r_1 und r_2 mit irgendwelchen Werten aus $|N_0|$ initialisiert, so kann der erste Befehl von Π ausgeführt werden und führt vom Zustand (a_1, r_1^0, r_2^0) - falls der erste Befehl kein HALTE-Befehl ist - zum Zustand (a_1, r_1^1, r_2^1) , d.h. $t_1(a_1, r_1^0, r_2^0) = (a_1, r_1^1, r_2^1)$. Ist der 1-te Befehl von Π kein HALTE-Befehl, so kann dieser wiederum ausgeführt werden usw. Ist $y = t_1 t_2 \dots t_m$ und (a_1, r_1^0, r_2^0) der Anfangszustand von Π , so wird statt

$$t_m(\dots t_2(t_1(a_1, r_1^0, r_2^0))\dots) = (a, r_1, r_2) \quad \text{auch}$$

$y(a_1, r_1^0, r_2^0) = (a, r_1, r_2)$ geschrieben, wobei a ein Element aus der Adreßmenge des Programms von Π ist.

2.2.18 Lemma

Sei Π eine Programm-Maschine mit einer beliebig vorgegebenen Initialisierung der Register r_1 und r_2 . Konstruiert

man eine Betriebsmittel-Maschine $\mathcal{L}$ nach den Vorschriften am Beginn des Beweises von Satz 2.2.13, so gilt folgende Aussage: Ist y irgendeine ausführbare Sequenz der Maschine Π , so gibt es eine ausführbare Sequenz x von $\mathcal{L}$, sodaß aus $y(a_1, r_1^\circ, r_2^\circ) = (a_1, r_1, r_2)$ und $x(q_0) = (x_1, x_2, z_1, \dots, z_n)$ folgt $x_1 = r_1$ und $x_2 = r_2$.

Beweis

Zu einer beliebigen ausführbaren Sequenz y von Π wird hier eine ausführbare Sequenz x von $\mathcal{L}$ rekursiv konstruiert, die die in diesem Satz erwähnte Eigenschaft besitzt. Mit y (bzw. x) werden in diesem Satz ausführbare Sequenzen von Π (bzw. $\mathcal{L}$) bezeichnet.

Die Aussage dieses Lemmas lautet:

$$\forall y \exists x \left[(y(a_1, r_1^\circ, r_2^\circ) = (a_1, r_1, r_2) \wedge x(q_0) = (x_1, x_2, z_1, \dots, z_n)) \Rightarrow (x_1 = r_1 \wedge x_2 = r_2) \right].$$

Hier wird nun folgendes bewiesen:

$$\forall y \exists x \left[(y(a_1, r_1^\circ, r_2^\circ) = (a_1, r_1, r_2) \wedge x(q_0) = (x_1, x_2, z_1, \dots, z_n)) \Rightarrow (x_1 = r_1 \wedge x_2 = r_2 \wedge |x| = |y| \wedge z_1 = 1 \wedge \forall j (j \neq 1 \Rightarrow z_j = 0)) \right]. \quad (*)$$

Dabei bedeutet $|y|$ bzw. $|x|$ die Länge der Sequenz y bzw. x . Der Beweis der zweiten Beziehung wird durch vollständige Induktion über der Länge von y durchgeführt.

a) Sei $|y| = 0$.

Dann ist $y = \Lambda$ (die leere Sequenz) und

$$\Lambda(a_1, r_1^\circ, r_2^\circ) = (a_1, r_1^\circ, r_2^\circ).$$

Für $x = \wedge$, also $|x| = |y|$ ist $x(q_0) = q_0 = (x_1^0, x_2^0, z_1^0, \dots, z_n^0)$.

Laut Initialisierung der Datenbasis von $\mathcal{L}$ am Beginn von Satz 2.2.12 folgt die Gültigkeit der rechten Seite der Beziehung (*).

b) Die Beziehung (*) gelte für alle y mit einer Länge $k (k \geq 0)$.

Behauptung: Die Beziehung (*) gilt auch für alle y mit der Länge $k+1$.

Beweis

Sei y eine ausführbare Sequenz von Befehlen der Maschine Π mit $|y| = k+1$ und zwar $y = t_1 t_2 \dots t_k t_{k+1} = y' t_{k+1}$.

Dann ist $y' = t_1 t_2 \dots t_k$, d.h. $|y'| = k$. Laut Induktionsannahme gibt es ein x' mit $x' = p_1 p_2 \dots p_k$, so daß für $y'(a_1, r_1^0, r_2^0) = (a_1, r_1, r_2)$ und $x'(q_0) = (x_1, x_2, z_1, \dots, z_n) = q$ gilt: $x_1 = r_1, x_2 = r_2, z_1 = 1$ und $\forall j (j \neq 1 \Rightarrow z_j = 0)$. Es ist also $y(a_1, r_1^0, r_2^0) = t_{k+1}(y'(a_1, r_1^0, r_2^0)) = t_{k+1}(a_1, r_1, r_2)$.

Aus der Konstruktion der Maschine Π folgt, daß der $k+1$ -te Befehl in der Sequenz y der i -te Befehl des Programms der Maschine Π ist. Weiterhin folgt, daß $i \neq n$ sein muß, d.h. der i -te Befehl kann nicht der HALTE-Befehl sein, da sonst y nicht ausführbar wäre. Man kann nun folgende Fälle unterscheiden:

b₁) t_{k+1} sei von der Art i r_1^+ mit $1 \in \{1, 2\}$

Dem i -ten Befehl des Programms von Π wurde laut Konstruktion der Elementar-Prozeß P_1 von $\mathcal{L}$ zugeordnet. Laut Induktionsannahme (siehe oben) gilt aber $x_1 = r_1, x_2 = r_2, z_1 = 1$ und $\forall j (j \neq 1 \Rightarrow z_j = 0)$.

Vom Zustand $x'(q_0) = (x_1, x_2, z_1, \dots, z_n)$ ist somit nur Elementar-Prozeß P_i von $\mathcal{L}$ laufbereit. Sei nun $P_i(q) = q' = (x'_1, x'_2, z'_1, \dots, z'_n)$ und $t_{k+1}(a_1, r_1, r_2) = (a_{i+1}, r'_1, r'_2)$.

Laut Konstruktion der Elementar-Prozesse von $\mathcal{L}$ ist $x'_1 = r'_1$, $x'_2 = r'_2$, $z'_{i+1} = 1$ und $\forall j (j \neq i+1 \Rightarrow z'_j = 0)$.

Schreibt man nun $x = x' p_{k+1} = p_1 p_2 \dots p_k p_{k+1}$, wobei p_{k+1} die Zustandsänderung darstellt, die durch einen Umlauf von P_i hervorgerufen wird, so ist der Beweis der Behauptung für diesen Fall erbracht.

$b_2)$ t_{k+1} sei von der Art i $r_1^- (a_m)$ mit $1 \in \{1, 2\}$.

$\beta_1)$ Ist nun $r_1 \geq 1$, so ist laut Induktionsannahme $z_1 = 1$, $\forall j (j \neq 1 \Rightarrow z_j = 0)$ und $x_1 \geq 1$. In diesem Fall ist sowohl P_{i+} und P_{i-} laufbereit. Nach Konstruktion der Elementar-Prozesse besitzt aber P_{i+} eine höhere Priorität als P_{i-} , sodaß P_{i+} in den Zustand "laufend" versetzt wird. P_{i-} wird zuerst in den Zustand "wartend" und nach einem Umlauf von P_{i+} in den Zustand "blockiert" versetzt werden. Bezeichnet man wiederum mit p_{k+1} die durch einen Umlauf von P_{i+} hervorgerufene Zustandsänderung, so folgt aus $p_{k+1}(q) = q' = (x'_1, x'_2, z'_1, \dots, z'_n)$, $t_{k+1}(a_1, r_1, r_2) = (a_{i+1}, r'_1, r'_2)$ aus der Induktionsannahme und der Konstruktion der Elementar-Prozesse von $\mathcal{L}$ wiederum die Gültigkeit der Behauptung.

β_2) Ist $r_1=0$, so folgt aufgrund ähnlicher Überlegungen wie in β_1), daß in diesem Fall der Elementar-Prozeß P_{i-} in den Zustand "laufend" versetzt werden kann, da der Prozeß P_{i+} nicht lafbereit ist. In diesem Fall wird mit P_{k+1} die durch die Ausführung von P_{i-} hervorgerufene Zustandsänderung bezeichnet. Auch in diesem Fall gilt offensichtlich die Behauptung der Induktion.

Somit ist Lemma 2.2.18 vollständig bewiesen.

2.2.19 Definition

Eine ausführbare Sequenz x von Prozeßumläufen der Maschine $\mathcal{L}$ heißt Verklemmungssequenz, wenn für $q=x(q_0)= (x_1, x_2, z_1, z_2, \dots, z_n)$ gilt: $\forall i (1 \leq i \leq n \Rightarrow z_i=0)$. Nach Ausführung einer Verklemmungssequenz ist $\mathcal{L}$ total verklemmt.

2.2.20 Definition

Eine ausführbare Sequenz y von Befehlen der Programm-Maschine Π heißt Haltesequenz, wenn gilt:

$y(a_1, r_1^0, r_2^0) = (a_n, r_1, r_2)$, wobei a_n die Adresse des HALTE-Befehls des Programms von Π ist.

Nach Ausführung einer Haltesequenz y von Π ist ein Zustand erreicht, von dem aus kein weiterer Befehl von Π ausgeführt werden kann.

Ist eine beliebige Programm-Maschine Π gegeben und bildet man nach den Konstruktionsvorschriften am Beginn des Satzes 2.2.13 die entsprechende Betriebsmittel-Maschine,

so wird diese Maschine der Einfachheit wegen mit $\mathcal{L}^*$ bezeichnet.

2.2.21 Lemma

Ist $y = t_1 t_2 \dots t_k$ eine ausführbare Sequenz von Π und $x = p_1 p_2 \dots p_k$, die nach Lemma 2.2.18 konstruierte ausführbare Sequenz von $\mathcal{L}^*$, so gilt: y ist genau dann eine Haltesequenz von Π , wenn $x' = x p_{k+1}$ eine Verklemmungssequenz von $\mathcal{L}^*$ ist.

Beweis

$\Rightarrow$ Sei $y = t_1 t_2 \dots t_k$ eine Haltesequenz von Π . Nach Definition 2.2.20 gilt: $y(a_1, r_1^0, r_2^0) = (a_n, r_1, r_2)$.

Aus dem Beweis von Lemma 2.2.18 folgt für die entsprechende Sequenz x von $\mathcal{L}^*$: Ist $q = x(q_0) = (x_1, x_2, z_1, \dots, z_n)$, so ist $z_n = 1$ und $z_i = 0$ für $1 \leq i < n$. Vom Zustand q von $\mathcal{L}^*$ ist also nur Elementar-Prozeß P_n laufbereit. Bezeichnet man mit p_{k+1} die durch einen Umlauf von P_n her-
vorgerufene Zustandsänderung, so ist $x' = x p_{k+1}$ ausführbar und für

$$\begin{aligned} q' &= x'(q_0) = x p_{k+1}(q_0) = p_{k+1}(x(q_0)) = p_{k+1}(q) = \\ &= (x'_1, x'_2, z'_1, \dots, z'_n) \end{aligned}$$

ist $z'_i = 0$ für $1 \leq i \leq n$, da P_n laut Konstruktion der Variablen z_n den Wert Null zuordnet und alle anderen Größen unverändert läßt. Die Sequenz $x' = x p_{k+1}$ ist also eine Verklemmungssequenz von $\mathcal{L}^*$.

$\Leftarrow$ Sei $x = p_1 p_2 \dots p_k p_{k+1}$ eine Verklemmungssequenz von $\mathcal{L}^*$.
 Laut Definition 2.2.19 gilt für $q = x(q_0) =$
 $= (x_1, x_2, z_1, \dots, z_n) \quad z_i = 0 \quad \text{für } 1 \leq i \leq n$. Da aber x aus-
 führbar ist, kann dies nur dann der Fall sein, wenn
 p_{k+1} der Ausführung von P_n der Maschine $\mathcal{L}^*$ entspricht.
 Für $x' = p_1 p_2 \dots p_k$ muß dann gelten:

$$x'(q_0) = (x_1, x_2, z_1, \dots, z_n) \Rightarrow (z_n = 1 \wedge \forall i (1 \leq i < n \Rightarrow z_i = 0)).$$

Aus dem Beweis von Lemma 2.2.18 folgt, daß auch je-
 der ausführbaren Sequenz $x = p_1 p_2 \dots p_k$ von $\mathcal{L}^*$ eine
 ausführbare Sequenz $y = t_1 t_2 \dots t_k$ von Π entspricht,
 falls p_k nicht dem Elementarprozeß P_n von $\mathcal{L}^*$ entspricht.
 Dies trifft zu bei der oberen Sequenz x' . Für die x'
 entsprechende Sequenz $y = t_1 t_2 \dots t_k$ muß somit gelten

$$y(a_1, r_1^0, r_2^0) = (a_n, r_1, r_2).$$

Sequenz y ist also eine Haltesequenz von Π .

Somit ist gezeigt, daß zu einer beliebig vorgegebenen
 Programm-Maschine Π eine Betriebsmittel-Maschine $\mathcal{L}^*$ ange-
 geben werden kann, sodaß $\mathcal{L}^*$ im Sinne der Definition 2.2.12
 die Maschine Π simuliert. Satz 2.2.13 ist somit voll-
 ständig bewiesen.

Aus diesem Satz lassen sich einige für die Praxis der
 Betriebssysteme sehr wichtige Aussagen ableiten. Am An-
 fang des Beweises von Satz 2.2.13 wurde eine Vorschrift
 bzw. ein Algorithmus angegeben, der zu einer beliebig
 vorgegebenen Programm-Maschine Π nach endlich vielen
 Schritten eine Betriebsmittel-Maschine $\mathcal{L}$ findet, die Π
 simuliert. Aus Lemma 2.2.21 kann man nun folgende Aus-
 sage ableiten.

2.2.22 Korollar

Es gibt einen Algorithmus, der für jede Programm-Maschine Π nach endlich vielen Schritten eine Betriebsmittel-Maschine $\mathcal{L}^*$ konstruiert, so daß Maschine Π genau dann anhält, wenn $\mathcal{L}^*$ in eine totale Systemverklemmung gerät.

2.2.23 Definition

Unter dem Problem der totalen Systemverklemmung für Betriebsmittel-Maschinen versteht man die Frage, ob eine beliebig vorgegebene Betriebsmittel-Maschine in eine totale Systemverklemmung geraten kann.

2.2.24 Definition

Unter dem Sicherheitsproblem von Betriebsmittel-Maschinen versteht man die Frage, ob eine beliebig vorgegebene Betriebsmittel-Maschine frei von totalen Systemverklemmungen ist.

Korollar 2.2.22 zeigt, daß das HALTE-(NICHTHALTE)Problem der Programm-Maschinen auf das Problem der totalen Systemverklemmung (Sicherheitsproblem) von Betriebsmittel-Maschinen reduzierbar ist. Analog zu den Turing-Maschinen kann man eine universale Betriebsmittel-Maschine konstruieren, die sich selbst und jede beliebig vorgegebene Betriebsmittel-Maschine simulieren kann. Dann ist es offensichtlich, daß das Problem der totalen Verklemmung partiell lösbar ist. Zusammenfassend kann man nun folgende Aussage ableiten.

2.2.25 Korollar

Das Problem der totalen Verklemmung von Betriebsmittel-Maschinen ist zwar nicht lösbar, aber partiell lösbar. Das Sicherheitsproblem von Betriebsmittel-Maschinen ist nicht partiell lösbar.

Die Aussagen über das Problem der totalen Verklemmung kann man bekanntlich folgendermaßen interpretieren: Es gibt keinen festen Algorithmus (Turing-Maschine), der beim Vorliegen einer beliebigen Betriebsmittel-Maschine $\mathcal{M}$ immer nach endlich vielen Schritten anhält und zwar mit der Antwort "Ja" (bzw. "Nein"), wenn $\mathcal{M}$ in eine (bzw. keine) totale Systemverklemmung geraten kann. Dafür gibt es aber einen festen Algorithmus, der sich beim Vorliegen einer beliebigen Betriebsmittel-Maschine $\mathcal{M}$ folgendermaßen verhält (partiell lösbar): 1) Kann $\mathcal{M}$ in eine totale Systemverklemmung geraten, so hält der Algorithmus nach endlich vielen Schritten an und zwar mit der Antwort "Ja". 2) Kann aber $\mathcal{M}$ in keine totale Systemverklemmung geraten, so ist es nicht sicher, ob der Algorithmus mit einem "Nein" als Antwort anhält oder aber nie terminiert.

Im folgenden werden einige Bemerkungen gebracht, die dem Verständnis der Beziehung zwischen Turing-, bzw. Programm-Maschinen und Betriebsmittel-Maschinen dienen sollen.

Eine Betriebsmittel-Maschine besteht aus einer endlichen Anzahl von Variablen, Betriebsmitteln und Prozessen. Unter den Prozessen bestehen wiederum - mit Rücksicht auf ein Betriebsmittel - gewisse Prioritätsverhältnisse. Die Betriebsmittel-Maschine wurde so konstruiert, daß die gleichzeitige Tätigkeit von mehreren Prozessen gestattet ist, falls dadurch keine Konfliktsituationen entstehen. Aus der Beschreibung der Arbeitsweise von $\mathcal{M}$ (Definition 2.2.9) und Satz 1.2.8 ist nun folgendes offensichtlich: Führt eine Maschine $\mathcal{M}$ die Operation "Beenden von $P_1, \dots, P_k$ " aus, so kann die dabei erzielte Zustands-

änderung auch dadurch erreicht werden, indem dieselben Prozesse hintereinander in einer beliebigen Reihenfolge ausgeführt werden. Auch die Operation "Starten von $P_1, \dots, P_K$ " kann durch sequentielle Arbeitsweise simuliert werden. Es ist also möglich, einen Simulator (Algorithmus) anzugeben, der auf einem klassischen von Neumann-Rechner ablaufen kann und sich folgendermaßen verhält: Führt eine beliebig vorgegebene Maschine eine Zustandsänderung durch, so wird der Simulator durch sequentielle Ausführung mehrerer Operationen dieselbe Zustandsänderung erreichen. Somit sind die Betriebsmittel-Maschinen bezüglich ihrer Berechnungsfähigkeit den Turing- bzw. Programm-Maschinen äquivalent und zwar im gleichen Sinne wie deterministische und nicht-deterministische Turing-Maschinen äquivalent sind. Geht man von der Church'schen These aus, nach der die Turing-Maschinen die allgemeinsten Berechnungsmaschinen sind, so muß dies auch für Betriebsmittel-Maschinen gelten. Die letzteren Maschinen wurden mit Rücksicht auf die praktischen Belange der Betriebssysteme entwickelt. Andererseits ist ein Betriebssystem auch ein Algorithmus, der eine Lösung eines Problems (Betriebssystem-Problems) darstellt. Aus den obigen Ausführungen folgt nun, daß es zu jedem lösbaaren Betriebssystem-Problem eine Betriebsmittel-Maschine gibt, die dieses Problem löst.

2.2.26 Korollar

Die hier entwickelte Betriebsmittel-Maschine bildet eine universale Basis für Betriebssysteme.

Im Rahmen der Betriebssysteme gibt es verschiedene Aufgaben bei denen von Systemen die Rede ist, die unendlich viele Prozesse enthalten. Die Betriebsmittel-Maschine besteht aber aus endlich vielen Prozessen, sodaß auf-

grund des letzten Korollars offensichtlich ist, daß jedes Betriebssystem durch eine endliche Anzahl von Elementarprozessen dargestellt werden kann.

Wie bereits erwähnt, liegt eine totale Systemverklemmung vor, wenn sich sämtliche Prozesse der Betriebsmittel-Maschine im Zustand "blockiert" befinden. Ist eine solche Situation zustande gekommen, so kann kein Prozeß mehr in den Zustand "laufend" versetzt werden. Nun können auch solche Zustände erreicht werden, von denen aus eine gewisse nicht leere Teilmenge der Menge der Prozesse der Betriebsmittel-Maschine nie mehr in den Zustand "laufend" versetzt werden kann. Diese Situation nennt man auch eine partielle Systemverklemmung. Ähnlich zur Definition 2.2.23 kann man nun das Problem der partiellen Systemverklemmung definieren, wofür folgende Aussage gilt:

2.2.27 Satz

Das Problem der partiellen Systemverklemmung ist nicht lösbar.

Beweis

Der Beweis ist sehr einfach. Die totale Systemverklemmung ist nämlich ein spezieller Fall der partiellen Systemverklemmung.

Ist $\bar{P}$ die Menge der Prozesse einer Betriebsmittel-Maschine und P' eine echte Teilmenge von $\bar{P}$, so besteht das Problem der echt partiellen Systemverklemmung darin, ob $\bar{P}-P'$ sicher ist und gleichzeitig P' in eine Verklemmung geraten kann.

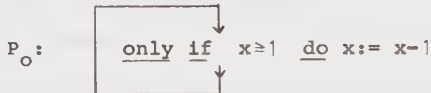
2.2.28 Satz

Das Problem der echt partiellen Systemverklemmung ist nicht partiell lösbar.

Beweis

Da dieser Satz in einem offensichtlichen Zusammenhang mit Satz 2.2.25 steht, wird hier der Beweis nur informell durchgeführt.

Sei $\mathcal{L}$ eine beliebig vorgegebene Betriebsmittel-Maschine mit der Prozeßmenge $\bar{P}$. Zu $\mathcal{L}$ kann man nun einen neuen Prozeß P_0 hinzufügen, wobei



gilt. Dabei soll P_0 so gewählt werden, daß $P_0 \notin \bar{P}$ ist und x nicht in der Menge der Variablen von $\mathcal{L}$ vorkommt. Außerdem soll x mit Null initialisiert werden. Die dadurch entstandene Maschine wird mit $\mathcal{L}'$ bezeichnet. Prozeß P_0 ist in $\mathcal{L}'$ auf jeden Fall verklemmt. Laut dieser Konstruktion kann $\mathcal{L}$ genau dann in keine totale Systemverklemmung geraten ($\mathcal{L}$ ist also sicher), wenn $\mathcal{L}'$ in eine echte partielle Verklemmung bezüglich $\{P_0\}$ geraten kann. Wäre nun das Problem der echt partiellen Verklemmung partiell lösbar, dann gäbe es einen festen Algorithmus A , der bei Anwendung auf eine beliebige Maschine $\mathcal{L}$ nach endlich vielen Schritten mit der Antwort "Ja" anhalten würde, falls $\mathcal{L}$ in eine echte partielle Verklemmung bezüglich einer Teilmenge $P' \subseteq \bar{P}$ geraten könnte. Dann wäre aber auch das Sicherheitsproblem partiell lösbar, denn nach obiger Konstruktion ist eine beliebige Maschine $\mathcal{L}$ genau dann sicher, wenn

die aus $\mathcal{L}$ konstruierte Maschine $\mathcal{L}'$ bezüglich $\{P_0\}$ in eine echte partielle Verklemmung geraten kann. Dies ist aber ein Widerspruch zum Korollar 2.2.25.

Somit sind einige für die Praxis der Betriebssysteme wichtige Aussagen abgeleitet worden. Das in diesem Abschnitt entwickelte Modell wurde zwar mit besonderer Rücksicht auf die praktischen Belange der Betriebssysteme entwickelt, es wurde aber möglichst einfach konstruiert. Dadurch wurde die Herleitung allgemeingültiger Aussagen vereinfacht. Im nächsten Abschnitt werden verschiedene Anwendungen dieser Maschine untersucht. Dabei wird die Gelegenheit ergriffen, um zu zeigen, wie man durch geeignete Zusammensetzungen zu einfacheren Programmen gelangen kann.

3. Beispiele und Anwendungen

In diesem Abschnitt werden einige für Betriebssysteme typische Probleme mit Hilfe von Betriebsmittel-Maschinen gelöst. Dabei wird die Gelegenheit aufgegriffen, um zu demonstrieren, wie man durch geeignete Konstruktionen gewissen physikalischen Sachverhalten Rechnung tragen kann.

3.1 Modell eines Auskunftssystems

Das Auskunftssystem wird hier durch einen gewissen Datenbestand dargestellt, auf den zwei Arten von Zugriffen erlaubt sind. Die Datenbank darf durch Lese-Operationen inspiziert und durch Schreib-Operationen modifiziert werden. Diese beiden Arten von Zugriffen auf die Datenbank können von einer beliebigen Anzahl von Individuen veranlaßt werden. Dabei werden allerdings folgende Einschränkungen gemacht:

- a) Auf der Datenbank darf zu jedem Zeitpunkt entweder eine beliebige Anzahl von Lese-Operationen oder eine einzige Schreib-Operation ausgeführt werden.
- b) Trifft zu einem gewissen Zeitpunkt t_0 der Wunsch für eine Schreib-Operation ein, so wird vom System folgendes Verhalten erwartet:
 - b_1) Wird auf der Datenbank zum Zeitpunkt t_0 keine Operation ausgeführt, so muß zum nächsten Zeitpunkt auf der Datenbank eine Schreib-Operation ausgeführt werden.
 - b_2) Wird zum Zeitpunkt t_0 eine Schreib-Operation ausgeführt, so muß nach Beendigung dieser Operation wieder eine Schreib-Operation auf der Datenbank ausgeführt werden.

- b₃) Wird zum Zeitpunkt t_0 eine Anzahl n ($n \geq 1$) von Lese-Operationen ausgeführt, so darf nach t_0 keine neue Lese-Operation mehr genehmigt werden. Nach Beendigung aller n Lese-Operationen darf die Schreib-Operation die Datenbank manipulieren.

Um einen Vergleich der in dieser Arbeit entwickelten Betriebssystem-Maschine mit anderen aus der Literatur bekannten Modellen zu ermöglichen, werden hier drei verschiedene Lösungsmöglichkeiten dieses Problems angegeben.

3.1.1 Eine Lösung mit Hilfe von Semaphoren und Zustandsgrößen.

Eine ausführliche Darstellung dieser Lösung findet man z.B. in [53]. An dieser Stelle wird nur auf das wesentlichste beschränkt.

```
integer nr, nw;
semaphore k1, k2, k3, w, r;
nr:=nw:=0; k1:=k2:=k3:=w:=r:=1;

READER: wait(k3);
        wait(r);
        wait(k1);
        nr:=nr+1;
        if nr=1 then wait(w);
        signal(k1);
        signal(r);
        signal(k3);
        read data;
        wait(k1);
        nr:=nr-1;
        if nr=0 then signal(w);
        signal(k1);
        Remainder of Reader;
        goto READER;
```

```
WRITER:  wait( $k_2$ );  
         nw:=nw+1;  
         if nw=1 then wait(r);  
         signal( $k_2$ );  
         wait(w);  
         modify data;  
         signal(w);  
         wait( $k_2$ );  
         nw:=nw-1  
  
         if nw=0 then signal(r);  
         signal( $k_2$ );  
         Remainder of Writer;  
         goto WRITER
```

3.1.2 Eine Lösung mit Hilfe bedingter kritischer Abschnitte

Die Record-Variable v enthält die Integer-Variablen rr und aw , während die Record-Variable w leer ist und dafür sorgt, daß zu jedem Zeitpunkt höchstens ein "write"-Operation stattfindet. Sowohl aw als auch rr werden mit Null initialisiert.

Writer	Reader
<u>with</u> v <u>do</u>	<u>with</u> v <u>when</u> $aw=0$ <u>do</u>
$aw:=aw+1$ <u>await</u> $rr=0$;	$rr:=rr+1$;
<u>with</u> w <u>do</u> "write";	"read";
<u>with</u> v <u>do</u>	<u>with</u> v <u>do</u>
$aw:=aw-1$;	$rr:=rr-1$;

3.1.3 Bemerkungen zu den beiden Lösungsmöglichkeiten

Es ist offensichtlich, daß die zweite Lösung viel übersichtlicher ist als die erste. Es ist bereits bekannt, daß es Probleme gibt, die unter alleiniger Verwendung von Semaphoren nicht gelöst werden können. Das nächste Beispiel aus [55] ist ein solches Problem.

Im Rahmen der vorliegenden Arbeit wurden exakte Beweise über die Grenzen der Leistungsfähigkeit von Systemen mit wait- und signal-Operationen gefunden. Hier wird aber nicht näher darauf eingegangen, da dies die Homogenität dieser Arbeit beeinträchtigen würde.

Die bedingten kritischen Abschnitte erlauben zwar eine übersichtliche Darstellung, sie weisen aber zwei wesentliche Nachteile auf:

- 1) Sie sind nicht so effizient, wie die Lösungen mit Hilfe von wait- und signal-Operationen ([53]).
- 2) Die bisher bekannten Algorithmen zur Implementierung dieses Konzeptes sind sehr aufwendig (z.B. [44]). Auch im Rahmen der vorliegenden Arbeit wurde versucht, diesen Aufwand zu reduzieren. Obwohl dies teilweise gelungen war, blieb der Aufwand immer noch relativ hoch. Von den selben Grundgedanken ausgegangen hat Schmid [79] eine Aufwandsreduzierung erreicht. Auch dieser reduzierte Aufwand dürfte als relativ hoch angesehen werden.

Bei den oberen Lösungen wurden Darstellungen benutzt, die in der Literatur üblich sind.

3.1.4 Eine Lösung mit Hilfe der hier entwickelten Betriebsmittel-Maschine

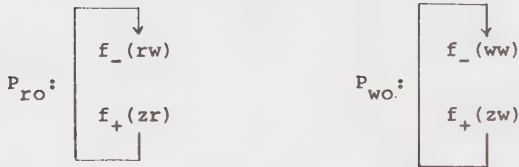
Es wird davon ausgegangen, daß es eine beliebige Anzahl von Terminals gibt, wobei jedes dieser Geräte einen

"Lese"-Knopf und einen "Schreibe"-Knopf besitzt. Zur Lösung dieser Aufgabe kann man dann zwei Modelle entwickeln:

a) Modell eines offenen Systems

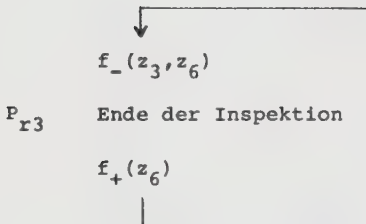
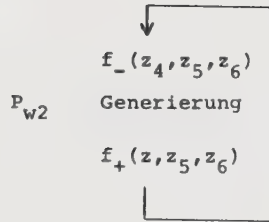
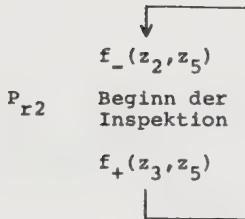
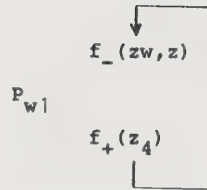
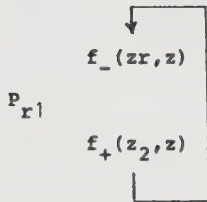
Die Terminals befinden sich außerhalb des eigentlichen Betriebssystemkerns. Sie können mit dem System über zwei Variablen (man könnte auch von Registern sprechen) rw und ww kommunizieren.

Die Variablen rw und ww mit $\text{Dom}(rw) = \text{Dom}(ww) = |N_0$ werden mit Null initialisiert. Bei jedem Druck auf den "Lese"- bzw. "Schreibe"-Knopf wird der Wert der Variablen rw bzw. ww um eins erhöht. Man führe zwei Elementarprozesse P_{ro} und P_{wo} ein, die zum Betriebssystem gehören und eine Interface-Funktion zwischen dem eigentlichen Betriebssystem und der aus beliebig vielen Terminals bestehenden Umgebung besitzen.



Aufgrund der speziellen Natur der Elementarprozesse können ohne Gefahr die Adressen der Prozesse weggelassen werden.

Die Betriebsmittel-Maschine enthalte weiterhin folgende Prozesse:



Die in dieser Betriebsmittel-Maschine vorkommenden Variablen werden in zwei Betriebsmittel eingeteilt. Die Variablen rw und ww seien im ersten, alle anderen im zweiten Betriebsmittel enthalten. Die Variablen z , z_5 und z_6 werden mit Eins, alle anderen mit Null initialisiert.

Die Prioritätsverhältnisse werden folgendermaßen definiert:

Im ersten Betriebsmittel ist die Prioritätsrelation leer.

Im zweiten Betriebsmittel gilt: $P_{w1} \succ P_{r1}$, $P_{r2} \succ P_{w2}$ und $P_{r3} \succ P_{w2}$.

Die Korrektheit dieser Maschine soll hier nur informell gezeigt werden.

Nach der Initialisierung des Systems befinden sich alle Prozesse der Maschine an ihrer Anfangsadresse und sind blockiert, so lange kein "Wunsch" zur Inspektion bzw. Generierung der Datenbank vorliegt. Man betrachte nun folgenden Ablauf des Systems:

Nach der Initialisierung seien k "Lese"-Wünsche eingetroffen. Prozeß P_{r0} erhöht den Wert von zr auf k . Nachdem nun Prozeß P_{r1} k_1 -mal ($k_1 \leq k$) gelaufen ist, treffe ein "Schreib"-Wunsch ein. Da Prozeß P_{w1} höhere Priorität besitzt als P_{r1} , wird nach der Erhöhung von zw auf Eins Prozeß P_{w1} vom Monitor des zweiten Betriebsmittels ausgewählt. Da Prozeß P_{w1} die Variable z auf Null setzt, bleibt P_{r1} mindestens solange blockiert, bis P_{w2} die Variable z wieder auf Eins setzt. Nachdem P_{r1} aber bereits k_1 -mal gelaufen ist, besitzt die Variable z_2 den Wert k_1 . Prozeß P_{w2} kann aber erst dann gewählt

und gestartet werden, wenn sowohl P_{r2} als auch P_{r3} insgesamt k_1 -mal gelaufen sind. Man nehme an, Prozeß P_{r2} sei insgesamt k_1' -mal gelaufen, wobei $k_1' < k_1$ ist. Prozeß P_{w2} befinde sich an seiner Anfangsadresse.

Befindet sich auch P_{r2} an seiner Anfangsadresse, so wird der Monitor 2 wegen der unterschiedlichen Prioritäten Prozeß P_{r2} starten. Befindet sich P_{r2} an seiner zweiten bzw. dritten Adresse, so kann P_{w2} nicht starten, da $z_5 = 0$ ist.

Ähnliche Beziehungen gelten zwischen P_{r3} und P_{w2} . Nachdem also sowohl P_{r2} als auch P_{r3} insgesamt k_1 -mal gelaufen sind, gilt:

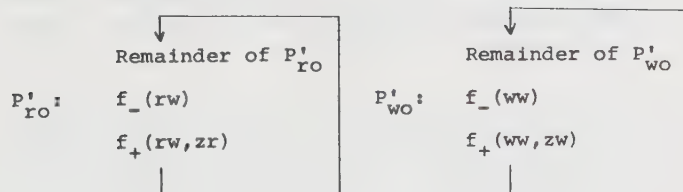
$$z = z_2 = z_3 = 0 \wedge z_4 = z_5 = z_6 = 1.$$

In diesem Zustand kann (mit Ausnahme der Interface-Prozesse) nur Prozeß P_{w2} gestartet werden. Danach wird Prozeß P_{w1} gestartet, falls $z_w \geq 1$ ist.

b) Modell eines geschlossenen Systems

Die Terminals gehören zum eigentlichen Betriebssystemkern.

In diesem Fall braucht man nur die Elementarprozesse P_{ro} und P_{wo} zu modifizieren.



Die Variablen rw und ww werden mit Eins initialisiert. Der Teil "Remainder of .." simuliert die Tatsache, daß die Zeitspanne zwischen zwei aufeinanderfolgenden Eintrittten von Lese- bzw. Schreibwünschen eine rein stochastische Größe ist. Dies eröffnet die Möglichkeit des Einsatzes von Wahrscheinlichkeitstheoretischen Ansätzen.

3.1.5 Bemerkungen zu diesen Lösungsmöglichkeiten

Die Tatsache, daß zu einem gewissen Zeitpunkt k "Lese"-Aktionen auf der Datenbank stattfinden, wird hier dadurch realisiert, daß zu diesem Zeitpunkt die Variable z_3 den Wert k besitzt. Besitzt zu einem Zeitpunkt die Variable rw bzw. ww den Wert 1, dann entspricht dies der Tatsache, daß zu diesem Zeitpunkt noch 1 nicht erfüllte Lese- bzw. Schreibwünsche vorliegen. Es könnte der Eindruck entstehen, daß der Parallelitätsgrad des Systems durch die endliche Anzahl der Prozesse eingeschränkt wird. Das Eintreffen von mehreren Lesewünschen z.B. aus beliebig vielen Terminals wird hier durch den sequentiell arbeitenden Prozeß P_{ro} erledigt. Dies ist eine Einschränkung, die nicht auf die Betriebsmittel-Maschine zurückzuführen ist. Wie auch immer man diese Tatsache realisieren will, es muß auf jeden Fall eine zentrale Stelle für die Entgegennahme der Lesewünsche eingerichtet werden und spätestens an dieser Stelle muß eine Warteschlange existieren, die nur seriell abgearbeitet werden kann.

Die hier angebotenen Lösungen könnten im ersten Augenblick ziemlich umfangreich erscheinen. Man sollte aber nicht vergessen, daß die hier entwickelten Systeme auch die Umgebung (Terminals) berücksichtigen. Außerdem bestehen diese Systeme nur aus Elementarprozessen.

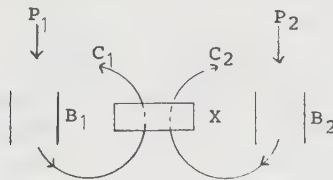
Durch die Bildung von zusammengesetzten Prozessen und durch die Einführung anderer Größen (Verträglichkeitsgrößen, Ereignisgrößen) kann man mit einer kleineren Anzahl von Variablen und Befehlen zurechtkommen. Dasselbe gilt für das nächste Beispiel.

3.2 Das zwei Erzeuger-, zwei Verbraucher-Problem

Dieses Problem kann bekanntlich (siehe 55) weder durch alleinige Verwendung von Semaphoren noch durch verallgemeinerte Petri-Netze, noch durch Vektor-Additions-Systeme bzw. Vektor-Substitutions-Systeme gelöst werden.

Das Problem sieht folgendermaßen aus:

Das System besteht aus zwei Erzeuger-Prozessen P_1 , P_2 und zwei Verbraucher-Prozessen C_1 , C_2 .

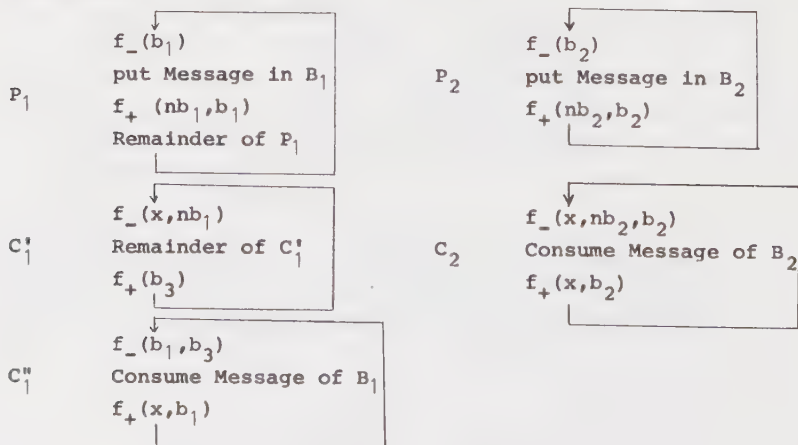


Zur Verfügung stehen außerdem zwei Puffer B_1 und B_2 mit unbeschränkter Kapazität und ein Gerät X , mit dessen Hilfe die Verbraucher C_1 bzw. C_2 auf B_1 bzw. B_2 zugrei-

fen können. Für das System gelten folgende Einschränkungen:

- Die Prozesse P_i und C_i dürfen nie gleichzeitig B_i manipulieren, wobei $i \in \{1, 2\}$ gilt.
- Die Prozesse C_1 und C_2 dürfen nie gleichzeitig das Gerät X benutzen.
- Ist B_i leer, so darf C_i keinesfalls auf B_i zugreifen.
- Anfangs sind beide Puffer leer. P_i erhöht die Anzahl der Informationen in B_i um Eins, C_i erniedrigt diese Zahl um Eins.
- Prozeß C_1 hat absolute Priorität gegenüber C_2 ; dies heißt, solange B_1 nicht leer ist, darf C_2 nie auf B_2 zugreifen.
- Sämtliche Prozesse dieses Systems sind zyklisch.

Die hier angebotene Lösung besteht aus fünf Prozessen. Prozesse P_1, P_2 und C_2 entsprechen den in der obigen Abbildung definierten Prozessen. Die Arbeitsweise des Prozesses C_1 wird hier durch die zwei Prozesse C_1' und C_1'' realisiert.



Mit den Variablen b_1, b_2 , wobei $b_1, b_2 \in \{0,1\}$, wird erreicht, daß die Puffer B_1 bzw. B_2 nur unter gegenseitigem Ausschluß von Prozessen P_1 und C_1^I bzw. P_2 und C_2 benutzt werden. Sowohl b_1 als auch b_2 werden mit Eins initialisiert. Die Variablen b_3, nb_1 (Anzahl der Nachrichten in B_1) und nb_2 (Anzahl der Nachrichten in B_2) werden mit Null initialisiert. Die Variable x wird mit Eins initialisiert und sorgt dafür, daß das Gerät X zu jedem Zeitpunkt von höchstens einem Prozeß benutzt wird.

An dieser Stelle braucht man nur zu zeigen, daß Forderung e) erfüllt ist; die Gültigkeit der übrigen Bedingungen ist nämlich offensichtlich.

Die Einschränkung e) kann man auch folgendermaßen formulieren: Ist Gerät X frei (d.h. $x=1$) und bewirbt sich Prozeß C_2 aus dem oberen Bild um X , dann darf C_2 nur dann X zugeteilt bekommen, wenn Puffer B_1 leer und Puffer B_2 nicht leer ist. Dies wird mit der angegebenen Maschine erreicht, wenn man vereinbart, daß Prozeß C_1^I höhere Priorität besitzt als C_2 . Falls $x=1$ ist, dann befinden sich die Prozesse C_1^I, C_1^I und C_2 an ihren Anfangsadressen. Laut Konstruktion der Elementarprozesse kann in diesem Falle Prozeß C_1^I nicht gestartet werden. Da C_1^I eine höhere Priorität als C_2 besitzt, wird zuerst geprüft, ob C_1^I gestartet werden kann. Dies ist genau dann der Fall, wenn $nb_1 \geq 1$ ist. Wird C_1^I gestartet, so kann danach C_2 auf keinen Fall gestartet werden, da x auf Null gesetzt wurde. Von der oben vorausgesetzten Ausgangssituation ($x=1$) kann Prozeß C_2 erst dann gestartet werden, wenn C_1^I nicht gestartet werden kann ($nb_1=0$) und $nb_2 \geq 1 \wedge b_2=1$ gilt. Dies heißt, daß in die-

sem Fall Puffer B_1 leer ist, während Puffer B_2 einerseits nichtleer ist ($nb_2 \geq 1$) und andererseits nicht vom Prozeß P_2 belegt ist ($b_2 = 1$). Die angegebene Betriebssystem-Maschine löst also die gestellte Aufgabe.

3.2.1 Bemerkungen zu diesem Beispiel

Im oberen Beispiel wurde stillschweigend ein fundamentaler, in der Praxis der Betriebssysteme häufiger und wichtiger Sachverhalt behandelt. Es gibt nämlich Synchronisationsaufgaben, in denen folgende Forderung vorkommt: Ein Prozeß P_i des Betriebssystems muß nur dann gestartet werden, wenn die von ihm benötigten Betriebsmittel verfügbar sind und außerdem die gesamte Datenbasis so aussieht, daß ein anderer Prozeß P_j ($i \neq j$) des Betriebssystems nicht gestartet werden kann. Hinter dieser Forderung steht die physikalische Tatsache, daß Prozeß P_j so eine Vorrangstellung innerhalb des Systems besitzt, daß in allen Fällen, in denen Prozeß P_j gestartet werden kann, keine Aktivitäten zugelassen werden, die eine eventuelle Verzögerung des Startens von Prozeß P_j hätten hervorrufen können. In der industriellen Regelung und Steuerung von Anlagen besitzen z.B. gewisse Alarmprozesse eine Vorrangstellung innerhalb des Prozeßsystems, da diese Prozesse die Anbahnung von Gefahren in der Anlage melden oder behandeln. In solchen Fällen muß die Reaktion auf die gefährliche Entwicklung sobald wie möglich erfolgen.

3.3 Alarmclock-Problem

Eine Anlage besitze eine Uhr, die in regelmäßigen Zeitabständen (virtuelle Zeiteinheiten) die Uhrzeit mitteilt. Ein Prozeß setzt sich zu einem Zeitpunkt t zur Ruhe und soll nach n virtuellen Zeiteinheiten wieder geweckt werden. Diese Situation kann durch eine Operation zweiter Art berücksichtigt werden, die sinngemäß folgendermaßen aussehen soll:

Ist die momentane Zeit $\geq$ Weckzeit, dann wecke Prozeß auf.

Die momentane Zeit wird durch die Uhr mitgeteilt. Die Weckzeit ist der Zeitpunkt, zu dem der blockierte Prozeß aufgeweckt werden soll. Stellt man die momentane Zeit durch eine Variable t und die Weckzeit durch eine Konstante wz dar, so wird die oben erwähnte Operation zweiter Art bei Erfüllung der Bedingung $t \geq wz$ den Wert der Variable t verändern. Hier wird also etwas mehr als erwünscht getan. Von der Aufgabenstellung her ist nämlich nur das Wecken des Prozesses zum Zeitpunkt wz erwünscht. Der selbe Sachverhalt liegt vor, wenn ein Prozeß in einem industriellen Regelungssystem die Grenzwertüberschreitung eines Meßwertgebers anmelden soll. All diesen physikalischen Sachverhalten kann man durch die Einführung von Ereignisvariablen (siehe weiter unten) Rechnung tragen. An dieser Stelle sei aber zu betonen, daß weder die Ereignisvariablen noch die Nichtverträglichkeitsrelationen (Abschnitt 3.4) die Leistungsfähigkeit der Betriebsmittel-Maschine erhöhen. Sie verbessern nur die Übersichtlichkeit der entsprechenden Programme und gestatten eine effizientere Implementierung.

Die Ereignisvariablen werden durch die Wirkung der Elementaroperationen zweiter Art auf diese Variablen charakterisiert. Die Vektor-Variable eines Betriebsmittels sei gleich $(v_1, v_2, \dots, v_l)$, wobei $v_1, v_2, \dots, v_k$ ($k \leq l$) Ereignisvariablen sind. Der feste c -Vektor (siehe Definition 2.2.1), mit dem eine Operation zweiter Art auf dieses Betriebsmittel zugreift, sei gleich $(c_1, c_2, \dots, c_l)$.

Diese Operation kann nur dann ausgeführt werden, wenn

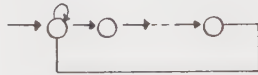
$$\forall i \ (1 \leq i \leq l \rightarrow v_i \geq c_i) \text{ gilt.}$$

Wird diese Operation ausgeführt, dann bleiben die Variablen $v_1, v_2, \dots, v_k$ (Ereignisvariablen) unverändert, während für die restlichen Variablen v_i ($k+1 \leq i \leq l$) die Zuweisung

$$v_i := v_i - c_i$$

stattfindet. Elementaroperationen erster Art auf Ereignisvariablen werden wie üblich ausgeführt.

Die bisher betrachteten Elementarprozesse besitzen ein äußerst einfaches Flußdiagramm:



Dabei bedeutet $\odot$ eine Operation zweiter Art und $\bigcirc$ eine Operation erster Art. Die Anzahl der Operationen erster Art kann unter Umständen auch Null sein. Man kann aber ohne weiteres komplizierte Flußdiagramme ableiten. Alle Prozesse, die ein anderes Flußdiagramm als die Elementarprozesse besitzen, werden Nichtelementarprozesse genannt. Hier werden zwei Beispiele angegeben.

Beispiel 1:

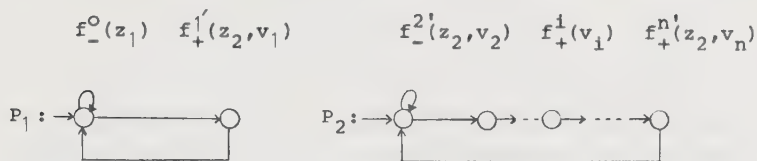
Gegeben sei ein Nichtelementarprozeß mit folgendem Flußdiagramm:

$$f_+^1(v_1) \quad f_-^2(v_2) \dots \quad f_+^i(v_i) \quad f_+^n(v_n)$$

P:



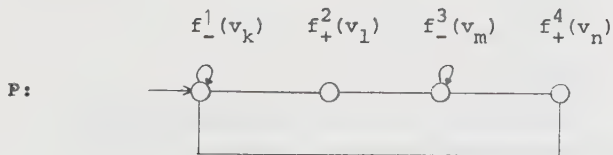
Die zweite Operation ist also eine Elementaroperation zweiter Art, während alle anderen Operationen erster Art sind. Man kann sich nun diesen Prozeß aus folgenden zwei Elementarprozessen zusammengesetzt denken:



In dem aus P_1 und P_2 bestehenden System werden die Variablen z_1 und z_2 eingeführt, die in dem Prozeß P nicht vorkommen. Dabei wird z_1 mit Eins, z_2 mit Null initialisiert. Nach obiger Abbildung wird eine neue Elementarfunktion zweiter Art f_-^0 eingeführt. Die Funktionen f_+^1 , $f_-^{2'}$ und f_+^n werden durch die Funktionen $f_+^{1'}$, $f_-^{2'}$ und $f_+^{n'}$ ersetzt, während alle anderen Funktionen f_+^i mit $2 \leq i < n$ in P und $\{P_1, P_2\}$ unverändert bleiben. Nun ist es offensichtlich, daß sowohl P als auch das aus P_1 und P_2 bestehende System bezüglich der Variablen $v_1, \dots, v_n$ die gleichen Berechnungen ausführen können.

Beispiel 2:

Das folgende Flußdiagramm entspricht einem Nichtelementar-Prozeß.



Dieser Prozeß kann durch folgendes System von Elementar-Prozessen ersetzt werden:

$$f_{-}^{1'}(z_1, v_k) \quad f_{+}^{2'}(z_2, v_1) \quad f_{-}^{3'}(z_2, v_m) \quad f_{+}^{4'}(z_1, v_n)$$



Dabei wurden zwei neue Variablen z_1 und z_2 eingeführt, wobei z_1 mit Eins und z_2 mit Null initialisiert werden. Die Funktionen f_{-}^1 , f_{+}^2 , f_{-}^3 und f_{+}^4 wurden in dieser Reihenfolge durch die Funktionen $f_{-}^{1'}$, $f_{+}^{2'}$, $f_{-}^{3'}$ und $f_{+}^{4'}$ ersetzt.

Beispiel 3:

Man betrachte den Elementarprozeß

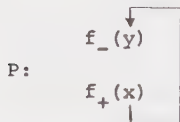


Dieser Prozeß wird genau dann blockiert, wenn $x=0$ wird, d.h. er realisiert die Wertzuweisung

$$x:=0.$$

Beispiel 4:

Der Elementarprozeß P mit

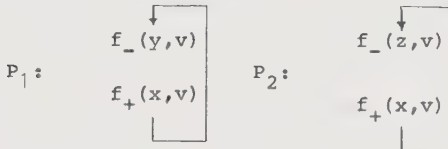


realisiert im allgemeinen die Wertzuweisung $x:=x+y$. Wird x mit Hilfe des vorigen Beispiels zuerst auf Null ge-

setzt, so kann man mit diesem Prozeß die Wertzuweisung $x:=y$ realisieren.

Beispiel 5:

Die aus P_1 und P_2 bestehende Maschine mit



realisiert die Wertzuweisung

$$x := y + z ,$$

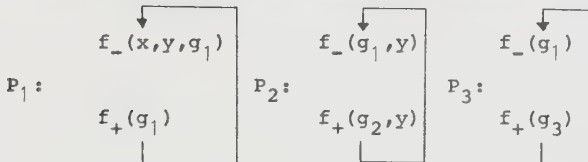
wenn sämtliche Variablen in einem Betriebsmittel zusammengefaßt werden, in dem P_1 eine höhere Priorität besitzt als P_2 und die Variable x mit Null, Variable v mit Eins initialisiert wird.

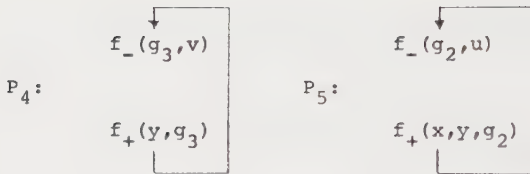
Beispiel 6:

Die Funktion

$$F(x, y) = \begin{cases} (x-y, y) & x \geq y \\ \text{falls} & \\ (x, y) & \text{sonst} \end{cases}$$

wird durch folgende Betriebsmittel-Maschine realisiert:





Sämtliche Variablen werden in einem Betriebsmittel zusammengefaßt. Für die Prozesse gilt $P_1 \succ P_2 \succ P_3$.

Die Variable u wird mit dem Wert der Variablen x und die Variable v mit dem Wert der Variablen y initialisiert. Weiterhin wird g_1 mit Eins, g_2 und g_3 mit Null initialisiert. Die Variablen x und y können mit beliebigen Werte aus $|N_0$ initialisiert werden.

Da P_1 die höchste Priorität besitzt, bleibt er aktiv, solange $x \geq 1$ und $y \geq 1$ gilt. Erst wenn P_1 blockiert wird, können überhaupt andere Prozesse gestartet werden. Nach seiner Blockierung hat P_1 die Funktion G realisiert, wobei

$$G(x, y, g_1) = \begin{cases} (x-y, 0, 1) & x \geq y \\ \text{falls} & \\ (0, y-x, 1) & \text{sonst} \end{cases}$$

Ist nach der Initialisierung $x \geq y$, so gilt nach der Blockierung von P_1 $y=0$. Danach kann nur Prozeß P_3 starten, der den Prozeß P_4 dazu veranlaßt, die Variable y auf ihren ursprünglichen Wert zu setzen.

Ist nach der Initialisierung $x \geq y$, so bewirkt P_1 nach seiner Blockierung die Zuweisung

$$x:=0 \text{ und } y:=y-x.$$

Da jetzt $y > 0$ ist und P_2 eine höhere Priorität besitzt als P_3 , wird nach der Blockierung von P_1 Prozeß P_2 gestartet, der wiederum den Prozeß P_5 veranlaßt, sowohl x als auch y auf ihre ursprünglichen Werte zu setzen. Nach der totalen Verklemmung der Betriebsmittel-Maschine ist die am Anfang dieses Beispiels gegebene Funktion F realisiert worden.

Man kann die Betriebsmittel-Maschine dahingehend erweitern, indem man für die Prozesse verschiedene Flußdiagramme zuläßt. Zusätzlich zu den Elementar-Operationen bzw. Funktionen nach Definition 2.2.1 kann man auch Funktionen nach den Beispielen dieses Abschnittes gebrauchen. Steht an der i -ten Stelle (Adresse) eines Prozesses die Operation

$$i \ F(x, y),$$

so kann man dafür in Anlehnung an die Elementar-Operationen zweiter Art folgende Interpretation und Implementierung geben. Kommt dieser Prozeß während einer Berechnung an seine i -te Adresse, so verhält er sich folgendermaßen: Ist $x \geq y$, so führt er die Zuweisung $x := x - y$ und wandert zur nächsten Adresse. Ist aber $x < y$, so wird dieser Prozeß blockiert und kann erst dann gestartet werden, wenn durch die Wirkung anderer Prozesse des Systems die Bedingung $x \geq y$ erfüllt ist. Natürlich müssen auch die Prioritätsverhältnisse beachtet werden. Definiert man aber x als Ereignisvariable und kommt der oben genannte Prozeß an seine i -te Adresse, so verhält er sich folgendermaßen: Ist $x \geq y$, so fährt der Prozeß weiter, ohne die Variablen x und y zu ändern. Ist aber $x < y$, so wird er blockiert.

Nun bietet das Alarmclock-Problem die Möglichkeit, diese Erweiterungen darzustellen. Zu diesem Problem werden hier zwei Lösungen angeboten.

Beispiel 7:

Erste Lösung des Alarmclock-Problems

Ein Prozeß P_i will sich zum Zeitpunkt t zur Ruhe setzen und soll nach n_i virtuellen Zeiteinheiten wieder geweckt werden. Für jeden solchen Prozeß $P_i (1 \leq i \leq k)$ wird eine Ereignisvariable t_i eingeführt und mit Null initialisiert. Für die Uhr wird ein Prozeß P_{Uhr} eingeführt, wobei

$$P_{Uhr}: f_+(t_1, \dots, t_k)$$

ist. Das Programmstück des Prozesses P_i mit der oben beschriebenen Funktion sieht dann folgendermaßen aus:

·
·
·
 $t_i := 0$

$P_i:$

only if $t_i \geq n_i$ do {
·
·
·

Es ist zu beachten, daß n_i eine Konstante ist. Mit { } wird hier die leere Anweisung dargestellt.

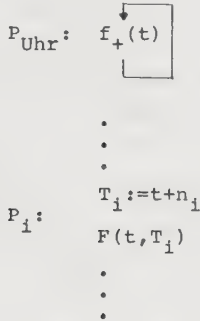
Beispiel 8:

Zweite Lösung des Alarmclock-Problems

Für sämtliche Prozesse des Systems wird eine Ereignisvariable t (virtuelle Zeit) eingeführt. Will sich ein Prozeß P_i zu einem Zeitpunkt t zur Ruhe setzen und erst nach n_i virtuellen Zeiteinheiten geweckt werden, so bildet er $T_i := t + n_i$. Nun ist T_i wiederum eine Variable.

In diesem Falle muß nun der Prozeß P_i an der entsprechenden Stelle die Abfrage $t \geq T_i$ stellen. Ist dies nicht erfüllt, so wird er blockiert. Ist aber diese Bedingung

erfüllt, so kann er weiterlaufen, ohne t zu verändern, da t eine Ereignisvariable ist. Hier handelt es sich also um die Funktion bzw. Operation $F(t, T_1)$ aus dem Beispiel 6. In Analogie zum Beispiel 7 kann man die Prozesse folgendermaßen gestalten:



Sowohl in diesem als auch im vorigen Beispiel ist f_+ eine Elementar-Operation erster Art.

Bemerkung

Um gewissen Problemen aus der Praxis der Betriebssysteme gerecht zu werden, kann folgende Vereinbarung getroffen werden: Zu jeder Ereignisvariablen x gibt es im System genau einen ausgezeichneten Prozeß ("Erzeuger-Prozeß" von x genannt), der die Ereignisvariable x als normale Variable behandelt. Insbesondere kann dieser Prozeß auf x eine Elementar-Operation zweiter Art nach Definition 2.2.1 ausführen.

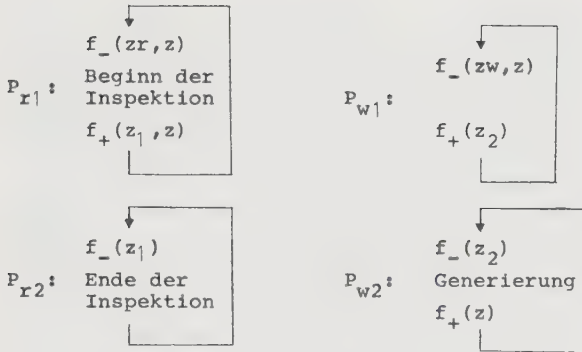
3.4 Nichtverträglichkeitsrelationen

Eine Nichtverträglichkeitsrelation $NVR \subseteq \bar{P} \times \bar{P}$ ist symmetrisch und nichtreflexiv. Dabei ist $\bar{P}$ die Menge der

Elementarprozesse einer Betriebsmittel-Maschine. Die Beziehung $(P_i, P_j) \in \text{NVR}$ (mit $i \neq j$) soll folgendes bedeuten: Hat P_i (bzw. P_j) seine erste Operation ausgeführt, so muß P_j (bzw. P_i) solange an seiner ersten Adresse warten, bis P_i (bzw. P_j) wieder an seiner ersten Adresse angekommen ist. Man kann auch ohne Nichtverträglichkeitsrelationen arbeiten. In diesem Falle braucht man für zwei sich nicht vertragende Prozesse P_i und P_j eine binäre Variable (Nichtverträglichkeitsvariable genannt), auf welche beide Prozesse mit ihren ersten und letzten Operationen zugreifen. Werden Nichtverträglichkeitsvariablen zur Bestimmung der Verträglichkeitsbeziehungen gebraucht, so bedeutet dies, daß die einzelnen Prozesse selbst die Entscheidungen treffen, mit welchen Prozessen sie sich nicht vertragen. Werden dagegen Nichtverträglichkeitsrelationen benutzt, so brauchen die einzelnen Prozesse nichts von den Verträglichkeitsbeziehungen untereinander zu wissen. In diesem Falle übernimmt der Monitor den damit verbundenen organisatorischen Aufwand. Die Bedeutung der beiden Auslegungen kann man einsehen, wenn man sich die einzelnen Prozesse einer Maschine als an einem Projekt beteiligte Personen vorstellt. Der Projektmanager hätte in diesem Fall die Kompetenzen eines Monitors.

3.4.1 Beispiel

Hier wird die Lösung 3.1.4 a) mit Hilfe einer einfachen Nichtverträglichkeitsrelation dargestellt. Die Interface-Prozesse P_{ro} und P_{wo} werden unverändert übernommen. Die neuen Prozesse sehen folgendermaßen aus:



Variable z wird mit Eins, alle anderen werden mit Null initialisiert. Die Variablen rw und ww werden wiederum im ersten, alle anderen im zweiten Betriebsmittel zusammengefaßt. Prozeß P_{r2} hat eine höhere Priorität als P_{w1} und P_{w2} , während P_{w1} eine höhere Priorität als P_{r1} hat. Nur die Prozesse P_{r2} und P_{w2} vertragen sich nicht.

3.5 Die Behandlung von Hardware-Einheiten mit einer Betriebsmittel-Maschine

Mit einer Variablen v_i einer Betriebsmittel-Maschine kann man die Anzahl der zur Verfügung stehenden Geräte vom Typ T_i darstellen. Ist die Zahl der in der Anlage vorhandenen Geräte vom Typ T_i gleich N_i ($N_i < \infty$), so gilt für den Wertebereich der Variablen v_i

$$\text{Dom}(v_i) = \{0, 1, \dots, N_i\}$$

Der Einfachheit wegen werden solche Variablen beschränkte Variablen genannt. Die bisherigen Arbeiten über Betriebssysteme haben sich fast ausschließlich auf Systeme mit

beschränkten Variablen konzentriert. In diesem Abschnitt werden Betriebsmittel-Maschinen betrachtet, die nur aus solchen Variablen bestehen. Aus dieser Interpretation folgt nun ohne weiteres, daß die in der Definition 2.2.1 eingeführten Elementaroperationen erster bzw. zweiter Art hier als "Freigeben" bzw. "Belegen" von Geräten interpretiert werden können, falls der momentane Wert einer Variablen v die Anzahl der momentan zur Verfügung stehenden Geräte des entsprechenden Typs wiedergibt. Nach der Initialisierung der Maschine stimmt Anzahl der zur Verfügung und Anzahl der vorhandenen Geräte jeden Typs überein. Deswegen ist jede Variable mit dem größten Wert ihres Wertebereiches zu initialisieren. Von einem Anfangszustand (Initialisierungszustand) einer Betriebsmittel-Maschine mit beschränkten Variablen können endlich viele Zustände erreicht werden. Somit sind die Fragen der Sicherheit und Verklemmung von solchen Maschinen entscheidbar. In diesem Abschnitt werden die Prioritätsrelationen außer acht gelassen. Es wird o.B.d.A angenommen, daß jeder Prozeß einer Maschine ein Flußdiagramm folgender Form besitzt:

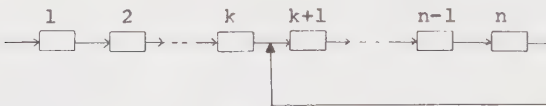


Bild 3.5.1 Flußdiagramm eines Prozesses

Dabei sollen die Rechtecke Elementaroperationen erster ("Freigabe" von Geräten) bzw. zweiter Art ("Belegen" von Geräten) symbolisieren. Mit den darüber liegenden Zahlen werden die Programm-Adressen des Prozesses bezeichnet. Nach der Initialisierung der Maschine wird

jeder Prozeß an seine erste Adresse angesetzt. Für Maschinen mit beschränkten Variablen ist es nun sinnvoll, gewisse Einschränkungen zu treffen, die ihre Rechtfertigung in der Art der physikalischen Systeme haben, die durch diese Maschinen simuliert werden. So kann die erste Operation in dem oberen Flußdiagramm nur eine Elementaroperation zweiter Art sein. Ein Prozeß kann nämlich nur Geräte freigeben, die er vorher belegt hat. Ist die Adressenmenge eines Prozesses P_i gleich $IA = \{1, 2, \dots, n\}$

so soll $z_i(v, k)$ die Anzahl der Geräte vom Typ T angeben, die Prozeß P_i nach Ausführung seiner k -ten Operation ($1 \leq k \leq n$) noch belegt hält. Vereinbarungsgemäß soll $z_i(v, 0)$ die Anzahl der vom Prozeß P_i belegten Geräte vom Typ T unmittelbar nach der Initialisierung des Systems darstellen. Es ist nun wiederum sinnvoll anzunehmen, daß für alle Variablen v und alle Prozesse P_i der Maschine gilt:

$$z_i(v, 0) = 0.$$

Die Funktion $z_i(v, k)$ kann nun folgendermaßen formal eingeführt werden:

3.5.1 Definition

Für alle Variablen v und alle Prozesse P_i einer Maschine gilt:

- a) $z_i(v, 0) = 0$
- b) Ist $IA = \{1, 2, \dots, n\}$ die Adressenmenge von P_i , so gilt für alle l mit $0 \leq l \leq n-1$:

$$z_i(v, l+1) = \begin{cases} z_i(v, l) & \text{falls } v \text{ im } (l+1)\text{-ten Befehl nicht vorkommt} \\ z_i(v, l) + m & \text{falls } v \text{ im } (l+1)\text{-ten Befehl um } m \text{ erniedrigt wird} \\ z_i(v, l) - m & \text{falls } v \text{ im } (l+1)\text{-ten Befehl um } m \text{ erhöht wird} \end{cases}$$

Sei nun P_i ein beliebiger Prozeß einer Maschine. An den Stellen k und n seines Flußdiagrammes nach Bild 3.5.1 gelten die Beziehungen

$$z_i(v, k) = K \text{ und } z_i(v, n) = L,$$

wobei v eine beliebige Variable der Maschine sei. Nun kann man zwischen drei Fällen unterscheiden:

a) Sei $K > L$.

Nach jedem Durchlauf entlang der Schleife von Bild 3.5.1 gibt P_i $K-L$ Geräte vom Typ T frei. Wird nach Initialisierung nur Prozeß P_i aktiv, dann kann er höchstens

$$\left\lfloor \frac{K}{K-L} \right\rfloor$$

mal entlang seiner Schleife im Bild 3.5.1 durchlaufen.

Hier versteht sich von selbst, daß ein Prozeß nur bereits belegte Geräte freigeben kann.

b) Sei $K < L$

Bei jedem Durchlauf entlang seiner Schleife belegt P_i $L-K$ Geräte vom Typ T mehr als im vorigen Durchlauf. Wird nach der Initialisierung des Systems nur Prozeß P_i aktiv, dann kann er höchstens

$$\left\lfloor \frac{N}{L-K} \right\rfloor$$

mal seine Schleife durchlaufen. Dabei ist N die größte Zahl aus dem Wertebereich von v . Ein Prozeß kann nämlich nicht mehr Geräte belegen, als im System vorhanden sind.

Beide oben angegebenen Zahlen sind obere Grenzen, die nicht unbedingt erreicht werden müssen.

c) Sei $K = L$

Ist für alle Variablen v und alle l mit $k \leq l \leq n$ $z_1(v, l) \leq N$ und wird nach der Initialisierung des Systems allein P_1 gestartet, so kann P_1 unendlich oft seine Schleife durchlaufen.

Es ist sinnvoll, wenn der Programmierer seine Prozesse so entwickelt, daß für alle Variablen v und alle Prozesse P_1 mit einem Flußdiagramm nach Bild 3.4.1 die Beziehung $K = L$ gilt. Außerdem sollte sich der Programmierer folgender Situation bewußt sein: Ist für alle l mit $k \leq l \leq n$ die Beziehung $z_1(v, l) \neq \emptyset$, so ist damit zu rechnen, daß Prozeß P_1 früher oder später eine feste Anzahl von Geräten des Typs T belegen und nie mehr freigegeben wird. Arbeitet das System ohne Verdrängung (was hier stillschweigend angenommen wurde), so ist dies mit Rücksicht auf die anderen Prozesse des Systems gewiß keine sinnvolle Art des Programmierens. Der Programmierer sollte dafür sorgen, daß es neben der Beziehung $K = L$ mindestens ein l mit $k \leq l \leq n$ gibt, sodaß für alle Variablen v die Beziehung $z_1(v, l) = 0$ gilt. Zusammenfassend kann man nun sagen, daß ein Programmierer, der seine Prozesse nach Bild 3.4.1 entwickelt, darauf zu achten hat, daß für alle Variablen v und alle Prozesse P_1 seines Systems gilt:

- 1) $z_1(v, k) = z_1(v, n)$
- 2) es gibt ein l mit $k \leq l \leq n$, sodaß $z_1(v, l) = 0$ ist,

3) für alle l mit $1 \leq l \leq n$ ist $z_l(v, l) \leq N$.

Hält sich der Programmierer an diese Regeln, so ist es sicher, daß es Möglichkeiten eines deadlockfreien Ablaufs des Systems gibt. Dies schließt die Möglichkeit von Verklemmungen nicht aus. Will man die Existenz von Verklemmungen in einer Maschine mit beschränkten Variablen vermeiden, so ist dies im allgemeinen mit einer schlechten Auslastung der Geräte des Systems verbunden. So ist es aus Effizienzgründen erwünscht, daß in solchen Maschinen Verklemmungszustände existieren. Durch dirigistische Maßnahmen soll aber vermieden werden, daß solche Zustände erreicht werden. Die aus der Literatur bekannten Algorithmen, die diese dirigistische Maßnahmen treffen, "laufen mit dem Betriebssystem mit". Vor Ausführung jeder Operation prüfen sie mit Hilfe irgendwelcher Kriterien, ob die Ausführung dieser Operation unbedingt zu einer Verklemmung führen wird. Ist dies der Fall, so wird diese Operation nicht ausgeführt. Der Schwerpunkt der aus der Literatur bekannten Arbeiten liegt in der Ermittlung von Algorithmen, die bei der oben genannten Überprüfung mit möglichst geringem Aufwand arbeiten. Hier wird eine Vorgehensweise vorgeschlagen, nach der der größte Teil der Untersuchungen ein einziges Mal, nämlich vor Inbetriebnahme des Betriebssystems, durchgeführt werden kann. Dies beruht auf der einfachen Tatsache, daß das Verklemmungs- und Sicherheitsproblem bei Betriebsmittel-Maschinen mit beschränkten Variablen entscheidbar sind.

3.5.2 Definition

Sei $\mathcal{L}$ eine beliebige Betriebsmittel-Maschine mit beschränkten Variablen. Ist q_0 ein Anfangszustand von $\mathcal{L}$,

so konstruiere man den Zustandsgraphen G von $\mathcal{L}$ nach folgender Vorschrift:

a) Ist K_n die Knotenmenge von G , so ist am Anfang

$$K_n = \{q_0\}.$$

b) Ist q ein Knoten von G , dann unterscheide man zwischen zwei Fällen:

b₁) Vom Zustand q aus kann kein Prozeß starten. Dieser Zustand ist ein totaler Verklemmungszustand der Maschine. Knoten q besitzt also keine Ausgangskanten.

b₂) Vom Knoten q aus können die Prozesse

$P_{i_1}, P_{i_2}, \dots, P_{i_k}$ ausgeführt werden. Dann führe

man k vom Knoten q ausgehende Kanten ein mit der Beschriftung $P_{i_1}, P_{i_2}, \dots, P_{i_k}$. Es gilt

$P_i(q) = q'$ genau dann, wenn die Ausführung von

P_i von q in q' überführt. Ist nun $P_{i_1}(q) \in K_n$

($1 \leq i \leq k$), dann endet die mit P_{i_1} beschriftete

Kante am Knoten $P_{i_1}(q)$. Ist $P_{i_1}(q) \notin K_n$, so er-

setze man K_n durch $K_n \cup \{P_{i_1}(q)\}$. Die mit P_{i_1} be-

schriftete Kante endet wiederum am Knoten $P_{i_1}(q)$.

Der so konstruierte gerichtete Graph ist endlich, da die Menge aller möglichen Zustände bei Betriebssystemen mit beschränkten Variablen endlich ist. Ein totaler Verklemmungszustand q in G ist dadurch gekennzeichnet, daß es von q aus keine auslaufende Kanten gibt. Ein Zustand q ist genau dann ein partieller Verklemmungszustand für P_i , wenn es keinen gerichteten Weg in G gibt, der von q zu einem Zustand q' von G führt, von dem aus P_i gestar-

tet werden kann. Sei q ein Verklemmungszustand des Systems nach folgendem Bild:



Gibt es Möglichkeiten eines deadlockfreien Ablaufs des Systems, so muß es auf dem gerichteten Weg s , der von q_0 nach q führt, mindestens einen Knoten geben, von dem man diesen Weg verlassen kann. Dann gibt es auch einen Knoten q' , für den gilt: Vom Zustand q' aus hat man zum letzten Male die Chance, den gerichteten Weg s zu verlassen. Tut man dies nicht, so wird man früher oder später an den Verklemmungszustand q geraten. Der gerichtete Weg von q' nach q wird "verbotener Weg", der Knoten q' "Entscheidungsknoten" genannt. Zu einem Verklemmungszustand kann es mehrere Entscheidungsknoten geben.

Wichtig ist an dieser Stelle die Tatsache, daß alle diese Feststellungen vor Inbetriebnahme des Systems getroffen werden können. Während des Ablaufs des Betriebssystems ist dann nur dafür zu sorgen, daß keine verbotenen Wege betreten werden. Dies kann dadurch geschehen, daß an allen Entscheidungsknoten die Ausführung solcher Prozesse verboten wird, die zu einem verbotenen Weg führen würden. Während des Ablaufs des Betriebssystems braucht man also nur zu fragen ob der momentane Zustand einem Entscheidungsknoten entspricht. Auf diese Weise ist es also möglich, einen wesentlichen Teil des Organisationsaufwandes vor Inbetriebnahme des Betriebssystems zu erledigen. Geht man nun davon aus, daß Betriebs-

systeme oft über eine längere Zeit, manchmal sogar über Jahre hinaus an einer Anlage laufen, dann wird durch die hier vorgeschlagene Vorgehensweise eine nicht unwesentliche Reduzierung des Verwaltungsaufwandes erreicht. Aus den vorangegangenen Erläuterungen ist die genauere Angabe der diesbezüglichen Algorithmen einfach, so daß hier darauf verzichtet werden kann.

4. Ein Konzept zur Implementierung der Synchronisation von Prozessen in einer Betriebsmittel-Maschine

Bei der Konstruktion von $\mathcal{L}$ (Abschnitt 2.2) wurde davon gesprochen, daß im Laufe einer Berechnung von $\mathcal{L}$ die einzelnen Prozesse ihre Zustände ändern. So war es unter anderem möglich, daß ein Prozeß vom Zustand "laufbereit" in den Zustand "laufend" bzw. "wartend" versetzt wird. Automatisch taucht nun die Frage auf, wie man diese Zustandsänderungen der Prozesse bei der Implementierung eines Betriebssystems realisieren kann. Aus Effizienzgründen ist es sinnvoll, von der Annahme auszugehen, daß einem blockierten Prozeß der Prozessor entzogen wird. Die Konsequenz dieser Vereinbarung ist nun die Tatsache, daß ein blockierter Prozeß keine Änderungen seines Zustandes mehr durchführen kann. Das Betriebssystem muß somit eine Instanz zur Verfügung stellen, die solche Änderungen übernimmt. Im Zusammenhang mit den Prioritätsrelationen bezüglich eines Betriebsmittels wurde bereits der Begriff "Monitor" erwähnt. Man kann ohne weiteres die Durchführung dieser Zustandsänderungen in den Kompetenzbereich des Monitors verlegen. Im Verlauf dieses Abschnittes wird dann ersichtlich, daß eine äußerst einfache und effiziente Implementierung der Synchronisation möglich ist. Zuerst aber wären einige Vorbemerkungen erforderlich.

Es gibt verschiedene Konzepte für die Synchronisation von mehreren Prozessen bzw. Aktivitäten in einem Betriebssystem. Die meisten davon kennen nur zwei Prozeßzustände, nämlich die Zustände "blockiert" und "laufend". Dies ist ein Spezialfall von Bild 2.2.8, in dem die Zustände "blockiert" und "wartend" im Zustand "blockiert", die

Zustände "laufend" und "laufbereit" im Zustand "laufend" zusammengefaßt werden. Bei dieser Zusammenfassung ist Vorsicht geboten. Es wurde öfter erwähnt und gezeigt, daß nicht sämtlich laufbereite Prozesse in den Zustand "laufend" versetzt werden können. Um einen Vergleich zwischen dem in diesem Kapitel zu entwickelnden Implementierungskonzept und den aus der Literatur bekannten Konzepten zu erleichtern, wird das Zustandsdiagramm von Bild 2.2.8 in der oben genannten Weise zusammengefaßt. Die wesentlichen Vorteile des hier entwickelten Konzeptes lassen sich auch für dieses Zustandsdiagramm demonstrieren. Eine dem Zustandsdiagramm von Bild 2.2.8 entsprechende Implementierung kann dann ohne weiteres abgeleitet werden.

In der Einführung wurden die Vor- und Nachteile der bekanntesten Synchronisationskonzepte (Monitor-Konzept und Konzept der bedingten kritischen Abschnitte) erwähnt. Der wichtigste Nachteil beider Konzepte war die Tatsache, daß die Ermittlung der Prozesse, die zu einem Zeitpunkt in den Zustand "laufend" versetzt werden können bzw. müssen, recht zeitaufwendig ist.

Laut Konstruktion von $\mathcal{L}$ können die Zugriffe auf verschiedene Betriebsmittel dieser Maschine völlig unabhängig voneinander stattfinden. Aus diesem Grunde wird zur Demonstration des Implementierungs-Konzepts nur ein Betriebsmittel v aus der Menge der Betriebsmittel von $\mathcal{L}$ betrachtet. Sämtliche Variablen einer Maschine $\mathcal{L}$, die keine Ereignisvariablen sind, werden hier Bedarfsvariablen genannt. Somit sind die Nichtverträglichkeitsvariablen (siehe 3.4) spezielle Bedarfsvariablen.

Sei nun $v=(v_1, v_2, \dots, v_k)$ ein Betriebsmittel, in dem $v_1, v_2, \dots, v_1$ ($1 \leq k$) Ereignisvariablen und $v_{1+1}, \dots, v_k$ Bedarfsvariablen sind. Da es mehr oder weniger eine Anschauungssache ist, ob man mit Nichtverträglichkeitsvariablen oder Nichtverträglichkeitsrelationen arbeitet, werden die zuletzt erwähnten Relationen zunächst außer Acht gelassen und erst später ins Konzept eingebaut. Weiterhin wird angenommen, daß $\mathcal{L}$ nur aus erlaubten Elementar-Operationen besteht.

Aus Abschnitt 2.2 kann ohne weiteres folgender, für die nächsten Ausführungen dieses Kapitels wichtiger Sachverhalt abgeleitet werden: Aus der Art einer Elementar-Operation, dem konstanten Vektor $(c_1, c_2, \dots, c_k)$ und der Art der Variablen eines Betriebsmittels $v=(v_1, v_2, \dots, v_k)$ kann die Wirkung dieser Operation ermittelt werden.

Sei $\{P_{i1}, P_{i2}, \dots, P_{in}\}$ die Menge der Elementar-Prozesse aus $\mathcal{L}$, die mit einer Operation zweiter Art auf $v=(v_1, v_2, \dots, v_k)$ zugreifen. Der Einfachheit wegen wird statt P_{i1} nur P_i geschrieben. Der feste c-Vektor, mit dem Prozeß P_i auf v zugreift, sei mit

$$c_i=(c_{i1}, c_{i2}, \dots, c_{ik})$$

bezeichnet. Die Matrix $C=(c_{ij})$ für $i=1, 2, \dots, n$ und $j=1, 2, \dots, k$ wird dann als Synchronisations-Matrix bezeichnet.

Kommt ein Elementar-Prozeß im Laufe einer Berechnung an die Stelle (Adresse) i seines Programms, an der er eine

Operation erster (zweiter) Art am Betriebsmittel v ausführen will, so meldet er sich an Q_{v+} (Q_{v-}) und wartet auf eine Genehmigung des Monitors M_v . Beide Warteschlangen können als verkettete Listen aufgefaßt werden, in denen die einzelnen Prozesse der Reihenfolge ihrer Anmeldung entsprechend eingetragen sind.

Unter den Elementen von Q_{v-} bestehen im allgemeinen gewisse Prioritätsverhältnisse, die unbedingt beachtet werden müssen. Aus diesem Grunde wird die Warteschlange K_v eingeführt. Jedes Element P_i aus Q_{v-} wird aus dieser Warteschlange entfernt und in die Warteschlange K_v gemäß folgender Vorschrift eingesetzt:

a) Ist K_v leer und wird P_j in K_v eingetragen, so enthält K_v nach dieser Eintragung nur P_j .

b) Besteht K_v aus den Elementen

$$P_{i_1} P_{i_2} \dots P_{i_{k-1}} P_{i_k} \dots P_{i_n}$$

in dieser Anordnung, so untersuche man die Elemente dieser Anordnung von links nach rechts.

b₁) Findet man einen Prozeß P_{i_k} , sodaß P_j eine höhere Priorität besitzt als P_{i_k} (d.h. $(P_j, P_{i_k}) \in PR_v$), dann plazierte man P_j vor P_{i_k} . Es ergibt sich die Anordnung $P_{i_1} P_{i_2} \dots P_{i_{k-1}} P_j P_{i_k} \dots P_{i_n}$.

b₂) Findet man kein solches Element, so plazierte man P_j am rechten Ende der Anordnung. Die neue Anordnung ist dann $P_{i_1} P_{i_2} \dots P_{i_{k-1}} P_{i_k} \dots P_{i_n} P_j$.

Diese Vorschrift ist eine rekursive Definition einer Operation, die hier "prioritätskonformes" Eintragen von Q_{V-} in K_V genannt wird und mit $Q_{V-} \rightarrow K_V$ bezeichnet wird. Unter den Elementen von K_V wurde eine totale Ordnung eingeführt, die die Prioritätsrelation PR_V nicht verletzt. Besitzt nämlich Prozeß P_i eine höhere Priorität als P_j und sind beide Prozesse in K_V enthalten, so liegt P_i immer vor P_j . Werden bei einer Ausführung der Operation $Q_{V-} \rightarrow K_V$ die Prozesse P_i und P_j ($i \neq j$) in K_V eingetragen und steht sowohl P_i als auch P_j in keinem Prioritätsverhältnis zu irgendeinem Prozeß aus K_V , so wird P_i genau dann vor P_j plaziert, wenn die Eintragung von P_i in Q_{V-} eher stattgefunden hat als die von P_j .

Die Synchronisations-Matrix C , die Warteschlangen Q_{V-} , Q_{V+} und K_V , sowie die Vektorvariable $(v_1, v_2, \dots, v_k)$, bilden die Datenbasis für die Synchronisation der Zugriffe auf das Betriebsmittel v .

Aus den bisherigen Ausführungen folgt nun, daß nach der Initialisierung die Warteschlangen Q_{V+} und K_V immer leer sind während Q_{V-} im allgemeinen aus mehreren Elementen besteht. Der Monitor eines Betriebsmittels kann erst nach Beendigung der Initialisierung des Systems anfangen zu "arbeiten". Im folgenden wird eine Möglichkeit zur Implementierung der Arbeitsweise des Monitors eines Betriebsmittels $v = (v_1, v_2, \dots, v_n)$ gegeben. An dieser Stelle sei aber nochmals betont, daß das hier entwickelte Konzept dem Programmierer einen großen Spielraum überläßt. Die einzigen Einschränkungen rühren von den Prioritätsrelationen her. Für Prozesse, die eine Operation erster Art auf v ausführen wollen, bestehen keine

prioritätsbedingten Einschränkungen. Da nun diese Operationen das System "positiv" beeinflussen, ist es sinnvoll, die Implementierung so zu programmieren, daß die Warteschlange Q_{V+} vor der Warteschlange Q_{V-} untersucht wird.

Der Algorithmus von Bild 4.1 gibt eine Möglichkeit für die Implementierung des Monitors M_V wieder. Zur Erläuterung dieses Flußdiagrammes werden zuerst die verschiedenen Aktivitäten des Monitors getrennt dargestellt.

- 1) Die Operation $Q_{V-} \rightarrow K_V$ wurde bereits erläutert. Der Monitor trägt die Elemente von Q_{V-} prioritätskonform in die Kette K_V ein. Danach ist Q_{V-} leer. Ist Q_{V-} vor Ausführung dieser Operation leer, so hat diese Operation keine Wirkung.
- 2) Bearbeitung von Q_{V+}

In der Liste Q_{V+} sind alle Prozesse enthalten, die gerade dabei sind, eine Operation erster Art auf $v=(v_1, v_2, \dots, v_k)$ auszuführen. Sind in Q_{V+} die Prozesse $P_1, P_2, \dots, P_n$ enthalten, so kann man u.a. folgende Implementierungsmöglichkeiten angeben:

- a) Nach Bild 4.2

Jeder Prozeß $P_i \in Q_{V+}$ enthält seinen festen c_i -Vektor $(c_{i1}, c_{i2}, \dots, c_{ik})$. Besitzt zu diesem Zeitpunkt die Vektorvariable $(v_1, v_2, \dots, v_k)$ des Betriebsmittels v einen gewissen Wert, so führt der Monitor M_V folgende Zuweisung aus:

$$(v_1, v_2, \dots, v_k) := (v_1, v_2, \dots, v_k) + \sum_{i=1}^n (c_{i1}, c_{i2}, \dots, c_{ik})$$

(Abkürzung: $\underline{v} := \underline{v} + \sum_{i=1}^n \underline{c}_i$).

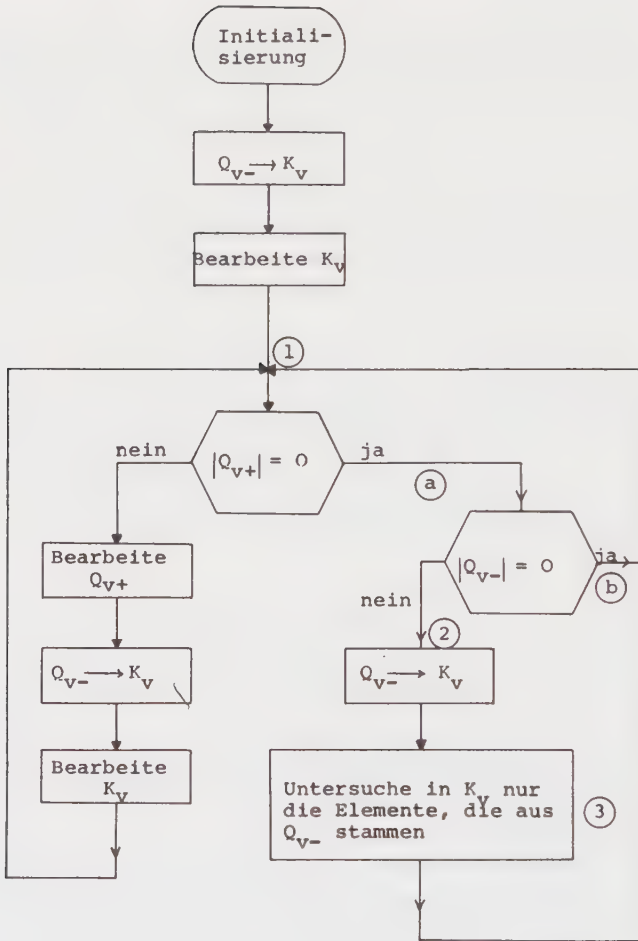


Bild 4.1: Arbeitsweise des Monitors M_V

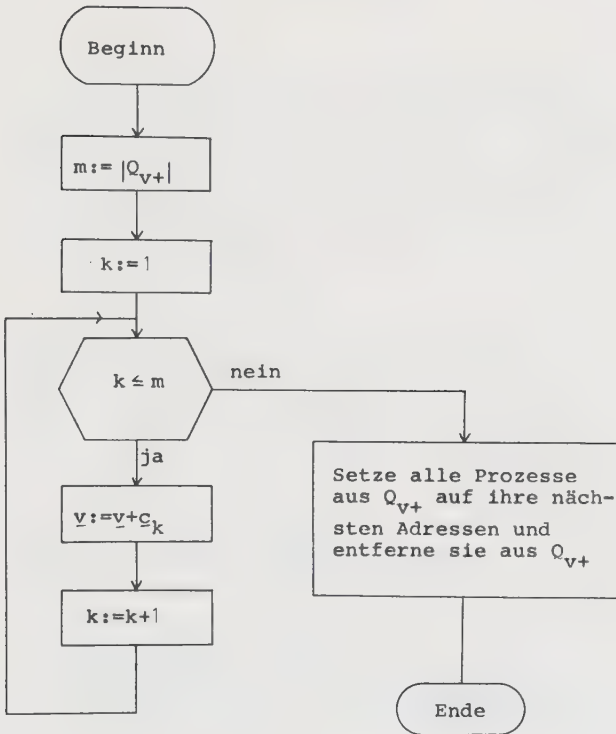


Bild 4.2: Bearbeitung von Q_{v+} nach 2a)

Danach wird jeder Prozeß aus Q_{v+} auf seine nächste Adresse gesetzt und aus dieser Liste entfernt.

b) Nach Bild 4.3

Der erste Prozeß aus Q_{v+} erhält vom Monitor die Genehmigung zur Ausführung seiner Operation und wird aus Q_{v+} entfernt. Der Monitor wartet auf die Fertigmeldung dieses Prozesses; danach wendet er sich dem zweiten Prozeß von Q_{v+} zu und verhält sich wie oben. Dies tut er solange bis Q_{v+} leer geworden ist.

Offensichtlich erspart man sich bei der ersten Implementierung den mit Umschaltvorgängen verbundenen organisatorischen Aufwand. Deswegen ist es doch sinnvoll, dem Monitor die Kompetenz zur Änderung der Zustandsvariablen eines Betriebsmittels zu übertragen.

3) Bearbeitung von K_v

In der Kette K_v sind alle Prozesse prioritätskonform eingetragen, die gerade dabei sind, eine Operation zweiter Art auf die Vektorvariable $(v_1, v_2, \dots, v_k)$ eines Betriebsmittels v auszuführen. Auch bei dieser Tätigkeit des Monitors kann man verschiedene Implementierungsmöglichkeiten angeben.

a) Nach Bild 4.4

Zuerst wendet sich der Monitor dem ersten Prozeß P_1 aus K_v zu. Mit Hilfe der Synchronisations-Matrix C kann er feststellen, ob $(v_1, v_2, \dots, v_k) \geq (c_{i1}, c_{i2}, \dots, c_{ik})$ gilt.

Ist dies nicht der Fall, so geht er zum nächsten Prozeß von K_v über. Ist aber diese Beziehung gültig, so verhält sich der Monitor analog zum Fall

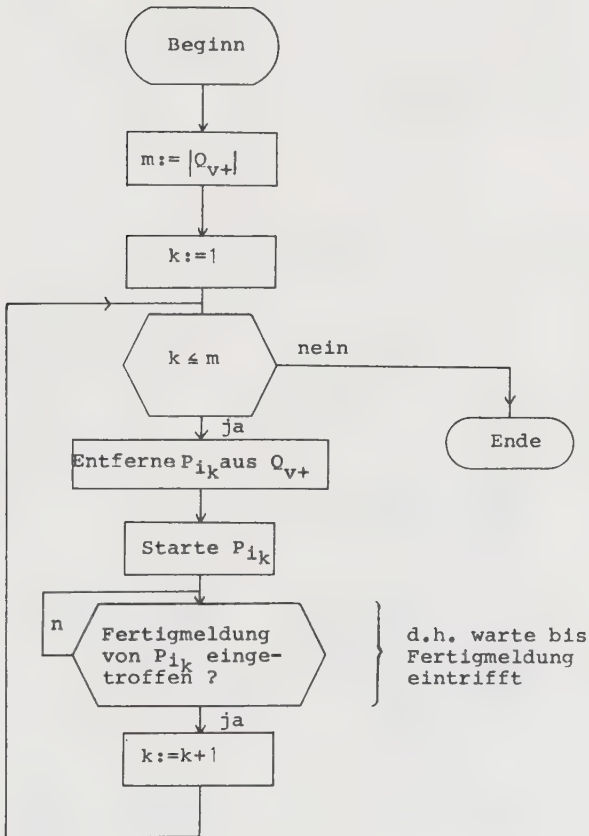


Bild 4.3: Bearbeitung von Q_{v+} nach 2b)

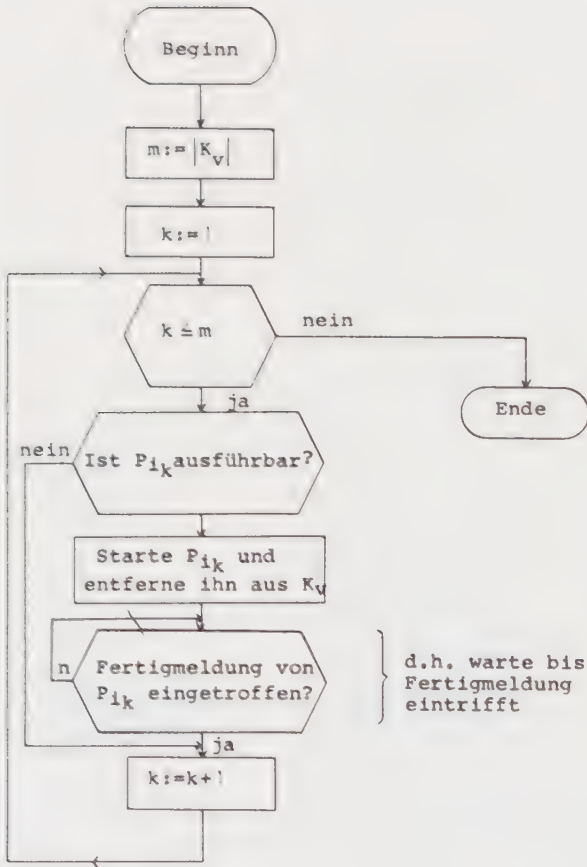


Bild 4.4: Bearbeitung von K_V nach 3a)

2b). Dieser Prozeß erhält vom Monitor die Genehmigung zur Ausführung seiner Operation und wird aus K_v entfernt. Nachdem der Monitor die Fertigmeldung dieses Prozesses erhalten hat, wendet er sich dem zweiten Prozeß aus K_v zu und verfährt analog. Dieser Vorgang wird solange wiederholt, bis sämtliche Prozesse aus K_v auf diese Weise untersucht worden sind.

b) Nach Bild 4.5

Der Monitor untersucht sämtliche Prozesse der Kette K_v nach der Reihenfolge ihrer Anordnung in K_v . Wie im ersten Fall kann der Monitor aus der Synchronisations-Matrix C die Startbedingung eines Prozesses P_1 aus K_v und die Variablenart des Betriebsmittels erkennen. Ist $(v_1, v_2, \dots, v_k) \triangleq (c_{11}, c_{12}, \dots, c_{1k})$, wobei $v_1, v_2, \dots, v_1$ Ereignisvariablen und alle anderen Bedarfsvariablen sind, so kann durch den Monitor die Zuweisung

$$(v_1, v_2, \dots, v_1, v_{1+1}, \dots, v_k) := (v_1, v_2, \dots, v_1, v_{1+1} - c_{11+1}, \dots, v_k - c_{1k})$$

ausgeführt werden. In diesem Falle wird der entsprechende Prozeß als "erledigt" bezeichnet. Nachdem sämtliche Prozesse aus K_v auf diese Weise untersucht worden sind, werden alle "erledigten" Prozesse aus K_v entfernt und erhalten vom Monitor die Genehmigung, mit ihren nächsten Adressen fortzufahren.

Hier gelten dieselben Aussagen, die in 2) gemacht wurden.

Nach diesen Erläuterungen kann die Arbeitsweise des Monitors nach Flußdiagramm 4.1 genauer beschrieben werden.

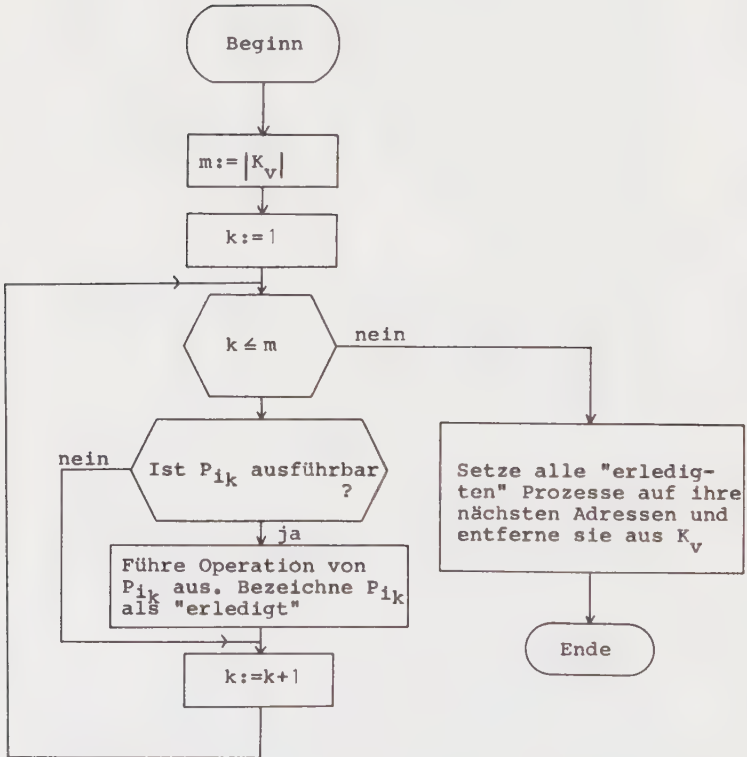


Bild 4.5: Bearbeitung von K_v nach 3b)

Wie bereits erwähnt wurde, sind nach der Initialisierung des Systems die Warteschlangen K_v und Q_{v+} eines Betriebsmittels v leer, während die Warteschlange Q_{v-} im allgemeinen aus mehreren Prozessen besteht. Unmittelbar nach der Initialisierung befindet sich nämlich jeder Prozeß an seiner ersten Adresse, an der laut Definition eine Elementar-Operation zweiter Art steht. Der Monitor trägt die Prozesse von Q_{v-} prioritätskonform in die Kette K_v ein ($Q_{v-} \rightarrow K_v$) und anschließend bearbeitet er K_v nach 3a) bzw. 3b). Befindet sich der Monitor an der Stelle ① seines Flußdiagramms (Bild 4.1), so wendet er sich zuerst der Warteschlange Q_{v+} zu. Laut Konstruktion der Betriebsmittel-Maschine unterliegen Prozesse, die eine Operation erster Art auf ein Betriebsmittel ausführen wollen, keinen prioritätsbedingten Einschränkungen. An der Stelle ① seines Flußdiagrammes könnte der Monitor irgendeine der Warteschlangen Q_{v+} und Q_{v-} als erste untersuchen. In der hier angegebenen Implementierung wendet sich der Monitor an der oben genannten Stelle zuerst der Warteschlange Q_{v+} zu, da die Prozesse aus Q_{v+} auf jeden Fall ausführbar sind und ihre Ausführung das System positiv beeinflusst. An dieser Stelle wendet sich der Monitor der Warteschlange Q_{v-} nur dann zu, wenn Q_{v+} leer ist. Ist neben Q_{v+} auch Q_{v-} leer, so läuft der Monitor so lange entlang der Kanten (a) und (b), bis eine Anmeldung an Q_{v+} oder Q_{v-} eintrifft. Dies ist eine Interpretationsmöglichkeit für die Tatsache, daß sich der Monitor im oben genannten Fall zur Ruhe setzt und erst dann aktiv wird, wenn eines der oberen Ereignisse eintrifft. Ist aber Q_{v+} leer und Q_{v-} nicht leer, so gelangt der Monitor von der Stelle ① zur Stelle ② seines Flußdiagrammes. Wegen dieser Voraussetzungen ist es klar, daß an dieser Stelle die Start-

bedingungen aller Prozesse aus K_V nicht erfüllt sind. Man braucht also nur die Startbedingungen der in Q_{V-} enthaltenen Prozesse zu untersuchen. Bei dieser Untersuchung müssen aber eventuell vorhandene Prioritätsverhältnisse unter den Prozessen von Q_{V-} unbedingt beachtet werden. Aus diesem Grund werden die Prozesse von Q_{V-} prioritätskonform in K_V eingetragen ($Q_{V-} \rightarrow K_V$).

An der Stelle ③ von Bild 4.1 braucht man nur die Startbedingungen der zuletzt in K_V eingetragenen Prozesse zu untersuchen. Zur Kennzeichnung dieser Prozesse in K_V ist ein Bit ausreichend. Man könnte natürlich auch sämtliche Prozesse in K_V untersuchen. Aus den obigen Erläuterungen ist aber klar, daß in diesem Fall unnötige Untersuchungen durchgeführt würden.

An dieser Stelle sei noch einmal betont, daß diese Beschreibungen Möglichkeiten zur Implementierung einer Betriebsmittel-Maschine darstellen.

Viele Einzelheiten der Konstruktion von Betriebssystemen wurden hier als bekannt vorausgesetzt. So wurde überhaupt nicht auf die Frage eingegangen, wie die Anmeldung eines Prozesses an eine Warteschlange zu realisieren ist. Auch die Tatsache, daß bei der Untersuchung einer Datenbasis gewisse Absperremechanismen einzusetzen sind, bzw. auf Absperrungen einer Datenbasis Rücksicht zu nehmen ist, wurde stillschweigend als bekannt vorausgesetzt. Es gibt verschiedene Dissertationen, deren Inhalt allein die ausführliche Implementierung eines Modells ist.

Gegenüber den aus der Literatur bekannten Implementierungen weist das hier entwickelte Konzept folgende Vorteile auf:

- 1) Für die Untersuchung einer Warteschlange Q sind hier höchstens $n = Q$ Überprüfungen notwendig. Bei den aus der Literatur bekannten Implementierungen ist folgender Fall möglich: Werden die Elemente einer Warteschlange Q der Reihe nach untersucht und ist z.B. der letzte Prozeß von Q startbereit, so wird er gestartet und aus Q entfernt. Anschließend muß gewartet werden, bis dieser Prozeß seine Operation beendet hat. Danach werden wiederum die Elemente der (neuen) Warteschlange der Reihe nach untersucht. Dieser Vorgang wird so oft wiederholt, bis entweder Q leer geworden ist oder bei einer Durchsuchung von Q kein Prozeß gestartet werden kann.
- 2) Aus der Synchronisations-Matrix, in der die festen c -Vektoren eingetragen sind, ist die Wirkung der Operation eines blockierten Prozesses ohne weiteres zu bestimmen. Somit braucht der Monitor nicht die ganze Operation zu kennen. Bei den üblichen Implementierungen besteht kein eindeutiger Zusammenhang zwischen der Bedingung und der Wirkung der Operation.
- 3) Da hier standardisierte Operationen und Datenarten benutzt werden, wäre die Entwicklung geeigneter Hardware (Mikroprogramme) zur Unterstützung der Software relativ einfach.

Arbeitet man ohne Nichtverträglichkeitsrelationen, so ist das Konzept der Implementierung an dieser Stelle als beendet zu betrachten. Die binären Nichtverträglichkeitsvariablen werden bei der Implementierung genauso behandelt, wie die Bedarfsvariablen. Arbeitet man aber mit Nichtverträglichkeitsrelationen, so müssen diese bei der Implementierung gesondert behandelt werden. Hier wird eine Möglichkeit dafür entwickelt.

Sei NVR_v eine nichtleere Nichtverträglichkeitsrelation bezüglich eines Betriebsmittels v gegeben. Laut Definition gilt: Ist $(P_i, P_j) \in NVR_v$, so ist $i \neq j$ und

$(P_j, P_i) \in NVR_v$. Sei $\{P_{i_1}, P_{i_2}, \dots, P_{i_k}\}$ die Menge der

Prozesse, die in mindestens einem der Paare $(P_1, P_m) \in NVR_v$ vorkommen. Der Einfachheit wegen wird statt P_{i_1} nur P_1 geschrieben. Man bilde den festen Spaltenvektor

$P = (P_1, P_2, \dots, P_k)$. Ist nun $(P_i, P_j) \in NVR_v$ und gilt $i < j$,

so führe man einen k -dimensionalen Spaltenvektor s_{ij} ein. Die i -te und j -te Komponente dieses Vektors werden mit Eins, alle anderen mit Null belegt. Weiterhin wird die binäre Variable α_{ij} eingeführt und mit Eins initialisiert. Aus der Symmetrie der Relation NVR_v folgt, daß die Anzahl der so konstruierten s_{ij} -Vektoren bzw. α_{ij} -

Variablen gleich $|NVR_v| : 2 = m$ ist. Die m Spaltenvektoren s_{ij} werden nebeneinander nach einer lexikographischen Anordnung der Indizes (ij) angeordnet, so daß eine $k \times m$ -Matrix entsteht. An dieser Matrix werden der k -dimensionale feste Vektor $P = (P_1, P_2, \dots, P_k)$ und der m -dimensionale variable Vektor $\alpha = (\alpha_{12}, \dots, \alpha_{ij}, \dots)$ nach Bild 4.6 angeordnet.

Die mit Null belegten Elemente dieser Matrix wurden leer gelassen. Jede Spalte enthält genau zwei Einsen, während eine Zeile auch mehrere Einsen enthalten kann. Ein Prozeß P_1 verträgt sich mit n Prozessen genau dann nicht, wenn in der 1-ten Zeile dieser Matrix n Einsen stehen. Jede Komponente des variablen Vektors α wird mit Eins initialisiert. Die i -te Zeile der oberen Matrix wird mit z_i bezeichnet. Der Elementar-Prozeß P_j befinde sich nun an seiner Anfangsadresse, an der er auf das Betriebs-

$\alpha =$	α_{12}	α_{13}	$\cdot \cdot \cdot \cdot$	α_{ij}	α_{ik}	α_{jk}	$\cdot \cdot \cdot \cdot$		
	s_{12}	s_{13}	$\cdot \cdot \cdot \cdot$	s_{ij}	s_{ik}	s_{jk}	$\cdot \cdot \cdot \cdot$		
P_1	1	1							← 1. Zeile
P_2	1								
$\vdots$									
P_i				1	1				
$\vdots$									
P_j				1		1			
$\vdots$									
P_k					1	1			← k. Zeile
	↑				↑				
	1. Spalte				m. Spalte				

Bild 4.6: Nichtverträglichkeits-Matrix

mittel v zugreifen will. Die Nichtverträglichkeitsrelation spielt für P_j nur dann eine Rolle, wenn es einen Prozeß P_1 gibt, so daß $(P_j, P_1) \in \text{NVR}_v$ gilt. In diesem Fall wird der Einfachheit wegen angenommen, daß P_j mit dem j -ten Prozeß in Bild 4.6 übereinstimmt. Von diesem Zustand aus kann P_j bezüglich NVR_v genau dann gestartet werden, wenn gilt $\alpha \geq z_j$. Wird dieser Prozeß vom Monitor dieses Betriebsmittels gestartet, so führt der Monitor die Operation $\alpha := \alpha - z_j$ aus. Mit der Beendigung der letzten Operation von P_j führt der Monitor die Operation $\alpha := \alpha + z_j$ aus.

Die Nichtverträglichkeits-Matrix wird somit genauso behandelt wie die Synchronisations-Matrix. Man kann nun durch Nebeneinanderreihen der beiden Matrizen und durch Erweitern des Zustandsvektors um den Vektor α eine erweiterte Synchronisations-Matrix und einen erweiterten Zustandsvektor erhalten. Mit der neuen Matrix und dem neuen Vektor verfährt der Monitor so, als gäbe es keine Nichtverträglichkeitsrelationen.

Als letztes sei nur noch eine Einzelheit bzw. Besonderheit erwähnt, die während der Implementierung Schwierigkeiten bereiten könnte.

Ein Betriebsmittel v bestünde aus den Variablen $(v_1, v_2, \dots, v_k)$, wobei alle v_i ($1 \leq i \leq k$) Nichtverträglichkeitsvariablen sind. Will nun der Programmierer alle Nichtverträglichkeitsvariablen streichen und nur entsprechende Relationen benutzen, so steht er vor dem Problem, daß jetzt das obere Betriebsmittel v keine Variablen mehr enthält. Eine erste Lösung dieses Problems bestünde darin, daß man mindestens eine der Variablen v_i ($1 \leq i \leq k$) im Betriebs-

mittel v beläst und als übliche Variable behandelt. Eine zweite Möglichkeit wäre die Einführung einer neuen Bezeichnung für die Elementar-Operationen erster und zweiter Art. Man könnte z.B. die Notation

allocate v only if....

für eine Elementar-Operation zweiter Art und

release v do....

für eine Elementar-Operation erster Art einführen. Enthält nun das Betriebsmittel v keine Variablen mehr, sondern nur eine Nichtverträglichkeitsrelation, so wird für die oberen Operationen einfach

allocate v und

release v

geschrieben.

Literatur

- [1] Agrewala, T.:
"A Complett Modell for Representing the
Coordination of Asynchronons Processes".
John Hopkins University, Baltimore (Md.),
Hopkins Computer Research Report 32(1974).
- [2] Anderson, J.P.:
"Program Structures for Parallel Processing".
CACM 8, 12(1965), 786-788
- [3] Baer, J.L.:
"A survey of Some Theoretical Aspects of
Multiprocessing".
Computing Surveys, 5, 1(1973), 31-80
- [4] Belpaire, G.; Wilmotte, J.P.:
"A semantic approach to the Theory of parallel
processes".
International Computing Symposium 1973,
North-Holland Publ. Co., 1974, 159-164
- [5] Brooks, N.B.:
"Jovial Automated Verification System (JAVS)".
General Research Corporation, Rome Air Deve-
lopment Center, Griffiss Air Force Base,
New York (1976), RADC-TR, 76-20
- [6] Bruno, J.L.; Coffman, E.G.; Hoaken, W.H.:
"Consistency of Synchronisation Nets using
P an V Operations".
The Pennsylvania State University, Computer
Science Department, Techn. Report No.117,
June 1972

- [7] Byrn, W.H.:
 "Sequential Processes, Deadlocks and Semaphore Primitives".
 Center for Research in Computer Technology,
 Harvard University, Cambridge, Mass. (1975),
 TR 7-75

- [8] Cohen, D.; Taft, E.:
 "A Parallel Processing Approach to Computer Communication".
 Center for Research in Computer Technology,
 Harvard University, Cambridge, Mass. (1973),
 TR 12-73

- [9] Conway, M.E.:
 "A Multiprocessor System Design".
 Proceedings-Fall Joint Computer Conference
 1963, 139-146

- [10] Coopridge, L.W.; Heymans, F.; Courtois, P.J.;
 Parnas, D.L.:
 "Information Streams Sharing a Finite Buffer:
 Other Solutions".
 Information Processing Letters, 3, 1(1974),
 16-21

- [11] Courtois, P.J.; Heymans, F.; Parnas, D.L.:
 "Concurrent Control with Readers and Writers".
 CACM 14, 10(1971), 667-668

- [12] Dennis, J.B.:
 "Programming Generality, Parallelism and
 Computer Architecture".
 Information Processing 68-North-Holland
 Publishing Comp.-Amsterdam (1969), 484-492

- [13] Dennis, J.B.; Van Horn, E.C.:
"Programming Semantics for Multiprogrammed Computations".
CACM 9, 3(1966), 143-155
- [14] Dijkstra, E.W.:
"The Structure of the "THE"-Multiprogramming System".
CACM 11, 5(1968), 341-346
- [15] Dijkstra, E.W.:
"Cooperating Sequential Processes".
In "Programming Languages" (Ed. F. Genuys),
Academic Press, London, 1968, 43-112
- [16] Dijkstra, E.W.:
"Hierarchical Ordering of Sequential Processes".
In "Operating Systems Techniques" (Ed. Hoare,
Perrott), Academic Press, London, 1972, 72-93
- [17] Genrich, H.J.; Lautenbach, K.:
"Synchronisationsgraphen".
Acta Informatica 2(1973), 143-161
- [18] Gilbert P.; Chandler, W.J.:
"Interference between Communicating Parallel Processes".
CACM 15, 6(1972), 427-437
- [19] Good, D.I.:
"An Interactive Program Verification System".
University of Southern California (1974),
ISI/RR-74-22

- [20] Grandoni, F.; Zerbetto, P.:
"Description and Asynchronous Implementation
of Control Structures for Concurrent Systems".
Internat. Computing Symposium 1973, North
Holland, Publ. Co., 1973, 151-158

- [21] Greif, I.:
"Semantics of Communicating Parallel Processes".
MIT, Project MAC(1975), Report Nr. MAC-TR-154,
Thesis.

- [22] Habermann, A.N.:
"Prevention of System Deadlocks".
CACM 11, 7(1969), 373-385

- [23] Habermann, A.N.:
"Synchronization of Communicating Processes".
CACM 15, 3(1972), 171-176

- [24] Habermann, A.N.:
"Path Expressions".
Carnegie-Mellon University, Pittsburgh (1975)
National Technical Information Service(1975),
AD-A015 842

- [25] Habermann, A.N.:
"A new Approach to Avoidance of System Dead-
locks".
Lecture Notes in Computer Science 16(1974),
163-170

- [26] Hack, M.:
"Decision Problems for Petri Nets and Vector
Addition Systems".
Computation Structures Group Memo 95, Project
MAC, MIT, March 1974

- [27] Hack, M.:
"The Recursive Equivalence of the Reachability Problem and the Liveness Problem for Petri Nets and Vector Addition Systems".
Automata and Switching Theory (1974), 156-164
- [28] Hansen, P.B.:
"The Nucleus of a Multiprogramming System".
CACM 13, 4(1970), 238-250
- [29] Hansen, P.B.:
"A Comparition of two Synchronizing Concepts".
Acta Informatica, 1(1972), 190-199
- [30] Hebalkar, P.G.:
"Deadlock-free Sharing of Resources in Asynchronous Systems".
MIT (1970), Project MAC, TR-75 (Thesis)
- [31] Hoare, C.A.R.:
"Towards a Theory of Parallel Programming".
In "Operating Systems Techniques" (Ed. Hoare, Perrott), Academic Press, London, 1972, 61-71
- [32] Hoare, C.A.R.:
"Monitors: An Operating System Structuring Concept".
Computer Science Department, Standford University (1973), Report No. STAN-CS-73-401
- [33] Hoare, C.A.R.:
"Parallel Programming: An axiomatic Approach".
Computer Science Department, Standford University (1973), Report No. STAN-CS-73-394

- [34] Hofmann, F.:
"Formale Behandlung Syntaktischer Funktionen".
Siemens Forsch.-u. Entwickl.-Ber. 3, 6 (1974),
341-347
- [35] Hofmann, F.:
"Prozeßmodelle aus der Sicht des Betriebs-
system-Entwurfs".
Kolloquium über Parallelismus in der Infor-
matik, Erlangen, 1976
- [36] Hofmann, F.:
"Verifikation von Betriebssystemen".
Vortrag im Informatik-Kolloquium in Dortmund,
1976
- [37] Hofmann, F.:
"Prozeßmodelle und Synchronisationsmecha-
nismen".
Vortrag im Informatik-Kolloquium in Bonn, 1976
- [38] Holt, A.; Commoner, F.:
"Events and Conditions".
Record of the project MAC conference on con-
current systems and parallel computations,
ACM, June 1970, 3-52
- [39] Holt, R.C.:
"Some Deadlock Properties of Computer Systems".
Computing Surveys 4, 3(1972), 179-196
- [40] Horning, J.J.; Randell, B.:
"Process Structuring".
Computing Surveys 5, 1(1973), 5-30

- [41] Irani, K.B.:
"A proposal for a Programming Language
for Parallel Processing Enviroments".
University of Michigan, Systems Engineering
Laboratory (1975), Report No. RADC-TR-75-63
- [42] Jensen, E.P.:
"Prospects for Automatic Verification of
Programs".
University of Michigan (1974), Report No.
RADC-TR-74-16
- [43] Johnson, R.T.; Fitzwater, D.R.:
"Verification of Process Structures of
Interacting Digital Systems".
Computer Science Department, The University
of Wisconsin (1973), Technical Report 193
- [44] Jürgens, J.:
"Synchronisation paralleler Prozesse anhand
von Zuständen".
Dissertation, TU München, 1973
- [45] Karp, R.M; Miller, R.E.:
"Properties of a Model for Parallel Compu-
tations: Determinacy, Termination, Queueing".
SIAM J. Appl. Math. 14, 6(1966), 1390-1411
- [46] Karp, R.M; Miller, R.E.:
"Parallel Program Schemata: A mathematical
Model for Parallel Computation".
Journal of Computer and System Sciences 3
(1969), 147-195

- [47] Keller, R.M.:
"Vector Replacement Systems: A Formalism for
Modeling Asynchronous Systems".
TR 117, Computer Science Laboratory, Princeton
University, Dec. 1972

- [48] Keller, R.M.:
"Parallel Program Schemata and Maximal
Parallelism, I. Fundamental Results".
Journal of the ACM 20, 3(1973), 514-537

- [49] Keller, R.M.:
"Parallel Program Schemata and Maximal
Parallelism, II. Construction of Closures".
Journal of the ACM 20, 4(1973), 667-710

- [50] Keller, R.M.:
"A Fundamental Theorem of Asynchronous
Parallel Computation".
Lecture Notes in Computer Science, 24
(1975), 102-112

- [51] Keller, R.M.:
"Formal Verification of Parallel Programs".
CACM 19, 7(1976), 371-384

- [52] Keramidis, S.; Bolch, G.:
"Betriebsprogrammierung I".
Skript einer Vorlesung von Prof. Dr. F. Hofmann,
Erlangen 1974

- [53] Keramidis, S.:
"Ein Beitrag zur Theorie der Prozeßsysteme".
IMMD-Berichte 7, 3(1974)

- [54] Keramidis, S.:
"Betriebsprogrammierung II"
Skript einer Vorlesung von Prof. Dr. F. Hofmann,
Erlangen, 1976
- [55] Kosaraju, S.R.:
"Limitations of Dijkstra's Semaphore Primitive
and Petri Nets".
ACM Operating Systems Review, 7(1973), 122-126
- [56] Lampson, B.W.:
"A Scheduling Philosophie for Multiprocessing
Systems".
Com. of ACM, 11, 5(1968), 347-360
- [57] Minsky, M.L.:
"Computation: Finite and Infinite Machines".
Prentice-Hall, Inc., Englewood Cliffs, N.J.
(1967)
- [58] Lanzarone, G.A; Ornaqhi, M.:
"Program Construction by Refinements Preserving
Correctness".
The Computer Journal, 18, 1(1976), 55-62
- [59] Lautenbach, K.:
"Exakte Bedingungen der Lebendigkeit für eine
Klasse von Petri-Netzen".
Berichte der GMD, Bonn 1973, Nr. 82
- [60] Lauenthal, M.S.:
"Verification of Programms Operating on
Structured Data".
MIT, Project MAC(1974), Report Nr.MAC-TR-124

- [61] Lipton, R.J.; Snyder, L.; Zalcstein, Y.:
"A Comparative Study of Models of Parallel
Computation".
Automata and Switching Theory (1974), 145-155
- [62] Lister, A.M; Maynard, K.J.:
"An Implementation of Monitors".
Software-Practice and Experience, Vol 6(1976),
377-385
- [63] Luconi, F.L.:
"Asynchronous Computational Structures".
MIT Project MAC-TR-49 (1968)
- [64] Misunas, D.P.:
"Performance Analysis of a Data-Flow Processor".
Project MAC, MIT (1975), Report No GSG Memo-128
- [65] Misunas, D.P.:
"Structure Processing in a Data-Flow Computer".
Project MAC, MIT(1975), Report No GSG Memo-129
- [66] Neuhold, E.J.:
"The Formal Sementics of Operating Systems".
International Computing Symposium 1973, North-
Holland Publ.Co., 1974, 127-133
- [67] Newton, G.:
"Proving Properties of Interacting Processes".
Acta Informatica 4(1975), 117-126
- [68] Parent, M.R.:
" The Control Graph Models: A Unified Approach
to Performance Evaluation".
Computing and Information Sciences, Case
Western Reserve University, (1974), Thesis

- [69] Parnas, D.L.:
"On a Solution to the Cigarette Smoker's
Problem (without conditional statements)".
Com. of ACM, 18, 3(1975), 181-183
- [70] Patil, S.S.:
"Limitations and Capabilities of Dijkstra's
Semaphore Primitives for Coordination among
Processes".
Project MAC, Computation Structure Group
MEMO 57 (1971)
- [71] Patil, S.S.:
"Coordination of Asynchronous Events".
Project MAC, MIT(1970), Thesis, Report No.
MAC-TR-72
- [72] Pettersen, O.:
"Synchronisation of Concurrent Processes".
Computer Science Department, Stanford Uni-
versity (1975), Report No. STAN-CS-75-502
- [73] Reifer, D.J.:
"A Structured Approach to Modelling Computer
Systems".
The Aerospace Corporation, El Segundo, Cali-
fornia (1974), Report No. SAMSO-TR-75-3
- [74] Rodrigues, J.E.:
"A Graph Model for Parallel Computation".
MIT, Project MAC(1969), Report No. MAC-TR-64

- [75] Rosen, B.K.:
"Tree-Manipulating Systems and Church-Rosser Theorems".
Journal of the ACM 20, 1(1973), 160-187
- [76] Rumbaugh, J.E.:
"A Parallel Asynchronous Computer Architecture for Data Flow Graphs".
Project MAC, MIT(1975), Report No. MAC-TR-150
- [77] Slutz, D.R.:
"The Flow Graph Schemata Model of Parallel Computation".
Technical Report TR-53, Project MAC, MIT,
Sept. 1968
- [78] Schmid, H.A.:
"An Approach to the Communication and Synchronisation of Processes".
Proc. International Computing Symposium 1973,
Amsterdam: North-Holland (1974), 165-171
- [79] Schmid, H.A.:
"On the Efficient Implementation of Conditional Critical Regions and the Construction of Monitors".
Acta Informatica 16, 6(1976), 227-249
- [80] Trauboth, H.:
"Proposal for Hierarchical Description of Software-Systems".
George C. Marshall Space Flight Center,
Alabama (1973), Report No. NASA TN D-7200

- [81] Van Leeuwen, J.:
"A Partial Solution to the Reachability
Problem for Vector Addition Systems".
6th Annual ACM Symposium on Theory of Com-
puting, May 1974, 303-309

- [82] Vantilborgh, H.; van Lamsweerde, A.:
"On an Extension of Dijkstra's Semaphore
Primitives".
Information Processing Letters 1 (1972),
North Holland Publ. Co., 181-186

- [83] Zeigler, P.:
"Towards a formal theory of modeling and
Simulation: Structure preserving morphism".
Journal of ACM, 19 (1972), 742-764

- [84] Zeyn, A.:
"Algorithmen zur Deadlock-Vermeidung".
Dissertation, Uni. Dortmund, Abt. Informatik,
1976, Bericht Nr. 22/76

L e b e n s l a u f

Ich bin am 21.1.1942 in Rodon, Kr. Florina, Griechenland geboren. Dort besuchte ich in den Jahren 1949 bis 1955 die Volksschule. Von 1955 bis 1961 war ich Schüler des 6. Knabengymnasiums in Thessaloniki. Im darauffolgenden Jahr befaßte ich mich eingehend mit der deutschen Sprache. Ab Wintersemester 1962 studierte ich dann Physik und Elektrotechnik an der Friedrich-Alexander-Universität Erlangen-Nürnberg. Im Herbst 1970 legte ich die Diplom-Prüfung in Physik mit Nebenfach Elektrotechnik ab. Von November 1970 bis Februar 1973 beschäftigte ich mich am Institut für Technische Chemie an der Universität Erlangen mit der Steuerung und Regelung von chemischen Anlagen mit Hilfe von Prozeßrechnern. Seit Februar 1973 bin ich als wissenschaftlicher Mitarbeiter am Institut für Mathematische Maschinen und Datenverarbeitung der Universität Erlangen-Nürnberg tätig.

Erlangen, den 14.9.1977

S. Kneemann

Bisher sind folgende Arbeitsberichte des Instituts für
Mathematische Maschinen und Datenverarbeitung erschienen :
=====

Band 1
=====

- NR. 1 Schmitt, A. :
Theorie der nichtdeterministischen und unvoll-
ständigen Mealy-Automaten.
(Januar 1967)
- NR. 2 Göring, W., Henning, W., Rasche, W. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 1: Übersicht.
(Juli 1967)
- NR. 3 Namneck, P. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 2: Über Verfahren zur syntaktischen Analyse von
Sätzen formaler Sprachen.
(Juli 1967)
- NR. 4 Eggers, B. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 3: Möglichkeiten der automatischen Erkennung
und selbständigen Korrektur von syntaktischen Fehlern
in formalen Sprachen.
(Januar 1968)
- NR. 5 Schneider, H. J. :
Programmierung numerischer Probleme.
(Oktober 1967)
- NR. 6 Wolf, F. :
Untersuchungen über den Vorhersagefehler bei
stochastischen Automaten.
(Februar 1968)
- NR. 7 Klar, R. :
Rechenautomaten 1.
(April 1968)
- NR. 8 Schmitt, A., Fehling, P. :
Rechenautomaten 2.
(November 1968)
- NR. 9 Weiss, S. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 4: Ein Programmsystem zum Testen von Mustererken-
nungsverfahren unter Verwendung eines Sichtgerätes.
(Juli 1968)
- NR. 10 Dokumentation über die Arbeit des Instituts in den
Jahren 1966 bis 1968.
(Januar 1969)

Band 2

=====

- NR. 1 Vollmar, R. :
Über einen Automaten mit Speicherband.
(Januar 1969)
- NR. 2 Händler, W. :
Design Philosophy for Computer Systems.
(März 1969)
- NR. 3 Brackmann, U., Händler, W., Lange, M., Rasche, W.
Wolf, H. P. :
Beiträge zur Kommunikation Mensch - Maschine.
(April 1969)
- NR. 4 Haferkorn, J., Schmitt, A., Seidel, R., Wolf, F. :
Rechnergestützter Unterricht.
(April 1969)
- NR. 5 Böhme, J. F., Göring, W. :
Zwei Beiträge zur Mustererkennung.
(August 1969)
- NR. 6 Köhl, J. :
Datenreduktion zur Speicherung ebener Figuren.
(April 1969)
- NR. 7 Haferkorn, J., Henning, W., Rasche, W. :
Ein bildschirmorientiertes Betriebssystem für die
PDP 7.
(Mai 1969)
- NR. 8 Saffert, J. :
Mikroprogramm-Steuerwerke.
(Juni 1969)
- NR. 9 Spies, P. P. :
Rechnerstrukturen und Ablaufgeschehen in Rechen-
automaten - Analysis und Simulation.
(September 1969)
- NR. 10 Händler, W., Wolf, F. :
Bericht über eine USA-Reise - Aktuelles aus dem
Gebiet der Informatik.
(September 1969)
- NR. 11 Dokumentation über die Arbeit des Instituts im
Jahre 1969.
(Januar 1970)

Band 3

=====

- NR. 1 Schmitt, A. :
Bildschirme und Kommunikation Mensch - Maschine.
(Februar 1970)
- NR. 2 Böhme, J. F. :
Experimente mit stochastischen Automaten.
(Juli 1970)
- NR. 3 Schneider, H. J. :
Chomsky-Systeme für partielle Ordnungen.
(August 1970)
- NR. 4 Haas, G. :
Programme zur dialogischen Logik.
(Oktober 1970)
- NR. 5 Weicker, R. :
Erkennung von kontextfreien Sprachen durch
Tabulator-Turingmaschinen.
(Juli 1970)
- NR. 6 Schmitt, A. :
Optimale Vorhersage der Ausgabe stochastischer
Automaten über lange Zeiten.
(Februar 1971)
- NR. 7 Dokumentation über die Arbeit des Instituts im
Jahre 1970.
(Februar 1971)

Band 4

=====

- NR. 1 König, R. :
Reduktion und Zerlegung von Automaten.
(Juni 1971)
- NR. 2 Klar, R. :
Messung von Rechneraktivitäten.
(Juli 1971)
- NR. 3 Augustin, U. :
A Note on Arbitrary Varying Channels with
Noiseless Feedback.
A Note on Channels with Finite Memory.
(August 1971)
- NR. 4 Brügmann, H. W. :
SIDIPRO, Dialogsprache und Echtzeit-Übersetzer
für die Programmierung eines Hochleistungs-
Sichtgerätes.
(August 1971)
- NR. 5 Händler, W. :
Strukturen von Computer-Graphics Systemen.
(Oktober 1971)
- NR. 6 Henning, W. :
Programmierung eines graphischen Sichtgerätes.
(April 1972)
- NR. 7 Dokumentation über die Arbeit des Instituts im
Jahre 1971.
(April 1972)

Band 5
=====

- NR. 1 Steierer, H. F. :
Algorithmen zur Zustandszerlegung von Automaten
unter besonderer Berücksichtigung der graphen-
theoretischen Aspekte.
(März 1972)
- NR. 2 Händler, W., Weizenbaum, J. :
Display Use for Man-Machine Dialog.
(September 1972)
- NR. 3 Dürre, K. :
Erfahrungen mit dem Programmiersystem LEKTOR-1
für Programmierten Unterricht.
(September 1972)
- NR. 4 Vollmar, R. :
SICELA - ein Interpretierer zur Simulation von
Automatennetzen.
(September 1972)
- NR. 5 Spies, P. P. :
Informatik III und Informatik IV.
Vorlesungsskriptum
(Oktober 1972)
- NR. 6 Görz, G. :
Die Sprache GEDANKEN als Beitrag zur syntaktischen
und semantischen Definition von Programmiersprachen.
(Oktober 1972)
- NR. 7 Buchzik, H. :
Maschinen-erzeugte Zufallszahlen.
(Dezember 1972)
- NR. 8 Dokumentation über die Arbeit des Instituts im
Jahre 1972.
(April 1973)

Band 6

=====

- NR. 1 Domesle, R. :
Vergleich von top-down- und bottom-up-
Kellerautomaten.
(Dezember 1972)
- NR. 2 Brüggmann, H. W. :
Zugriffsstrategien bei Rotationsspeichern.
(Juli 1973)
- NR. 3 Fachgespräch über mehrdimensionale Sprachen.
Erlangen, 14.-15. Juni 1973
- NR. 4 Händler, W. :
Bericht über einen USA-Aufenthalt 1972/1973
Aktuelles aus dem Gebiet der Informatik.
The Concept of Macropipelining.
(Mai 1973)
- NR. 5 Müller, H. :
Diskrete algebraische Strukturen.
(September 1973)
- NR. 6 Müller, H. :
Programmschemata mit Prozeduren.
(Oktober 1973)
- NR. 7 Leeb, K. :
Pascal Theorie.
(Dezember 1973)
- NR. 8 Chatterji, B. N. :
Two-stage Reinforcement Learning System.
An Algorithm for Economical Storage in Dynamic
Programming Approach to Pattern Recognition.
(Dezember 1973)
- NR. 9 Dokumentation über die Arbeit des Instituts im
Jahre 1973.
(Januar 1974)

Band 7

=====

- NR. 1 Eichmeier, P., Goeb, H. :
Anwendungsbeispiele von H-Graphen und VDL
zur semantischen Beschreibung.
(April 1974)
- NR. 2 Händler, W. :
Unconventional Computation by Conventional
Equipment.
(Mai 1974)
- NR. 3 Keramidis, S. :
Ein Beitrag zur Theorie der Prozesssysteme.
(August 1974).
- NR. 4 Nagl, M. :
Formale Sprachen von markierten Graphen.
(August 1974).
- NR. 5 Göttler, H. :
ALGOL 68 - Eine exemplarische Einführung.
(November 1974)
- NR. 6 Altmann, W. :
Konstruktionen zur Erweiterbarkeit von Sprachen.
(November 1974)
- NR. 7 Bunke, H. :
Beschreibung eines syntaxgesteuerten inkrementellen
Compilers durch Graph-Grammatiken.
(Dezember 1974)
- NR. 8 Jahresbericht 1974 der Informatik.
(März 1975)

Band 8

=====

- NR. 1 Ehrig, H., Nagl, M., Schneider, H. J. :
Three Papers on Graph Rewriting Systems and their
Applications.
(Januar 1975)
- NR. 2 Müller, H. :
Diskrete algebraische Strukturen II.
(April 1975).
- NR. 3 Klar, R., Wichmann, H. :
Mikroprogrammierung.
(Juli 1975)
- NR. 4 Weicker, R. :
Turingmaschinen mit assoziativem Speicherzugriff.
(Juli 1975)
- NR. 5 Schneider, H. J. :
Lexikon zur Programmierung I (ALGOL 60).
(Juli 1975)
- NR. 6 Woitok, R. :
Ein neuer Interpreter-Algorithmus für LISP
und seine Korrektheit.
(Juli 1975)
- NR. 7 Luft, A. :
Ein auf der Floyd/Hoare'schen Zusicherungsmethode
aufbauendes Verifikationsverfahren und seine
Begründung.
(August 1975)
- NR. 8 Rohrer, H. :
A Supervised Network of Adaptive Automata for
Pattern Recognition.
Drewes, B. :
Implementation of a Structural Pattern Recognition
System.
(Dezember 1975)
- NR. 9 Klar, R., Schreiber, H. :
Messungen mit dem ZÄHLMONITOR II.
(Dezember 1975)
- NR. 10 Jahresbericht 1975 der Informatik.
(März 1976)

Pand 9

=====

- NR. 1 Brendel, W., Frühauf, T., Nagl, M. :
Graph Rewriting Systems II.
Three Papers on Theory, Application, and Implementation.
(Februar 1976)
- NR. 2 Brügmann, H. W., Rudert, A., Lehr, H. :
EBIS
Strukturierte Bildprogrammierung auf höherer Ebene.
(Februar 1976)
- NR. 3 Hofmann, W., Wichmann, H. :
Konzept der Vertikalverarbeitung.
Implementierung und Analyse vertikaler Algorithmen
auf einem mikroprogrammierten Rechner.
(Mai 1976)
- NR. 4 Jordan, H. F., Smith, B. J. :
The Hardware-Software Interface.
(Juni 1976)
- NR. 5 Gernoth, B., Schmidt, B. :
Das Modell MEM-SIM.
Beschreibung eines Simulationsmodells zur Unter-
suchung einer Rechenanlage mit einer symmetrischen
Mehrprozessorkonfiguration und privaten Speichern.
(Oktober 1976)
- NR. 6 Grötsch, E. :
Vergleichende Studie über parallele Ersetzung
markierter Graphen.
Jahn, B. :
Eine Studie zur Implementierung von Graph-Grammatiken
unter Verwendung von Assoziativverarbeitung.
(Dezember 1976)
- NR. 7 Mertens, P., Neuwirth, W., Purucker, G.,
Schaller, K., Schmitt, W. :
Verknüpfung von Daten- und Methodenbanken, dargestellt
am Beispiel der Analyse von Marktforschungsdaten.
(Dezember 1976)
- NR. 8 Parallelismus in der Informatik.
Wissenschaftliches Kolloquium aus Anlass des Zehnjährigen
Bestehens des IMMD, Erlangen, 10./11. Juni 1976.
(Dezember 1976)
- NR. 9 Jahresbericht 1976 der Informatik.
(März 1977)

Band 10

=====

- NR. 1 Hoffmann, W. :
Modelle und Analysen von Teilnehmerrechensystemen,
angewendet auf das Teilnehmersystems RESPOND unter
CD 3300 MASTER.
(Februar 1977)
- NR. 2 Körber, K., Werzinger, G. :
Das Modell SIM-LAB.
Simulation eines klinisch-chemischen Laboratoriums.
(März 1977)
- NR. 3 Altmann, W., Weber, D. :
Programmiermethodik.
(März 1977)
- NR. 4 Maehle, E. :
Fehlertolerante Rechnerstrukturen.
(April 1977)
- NR. 5 Beth, T. :
Kombinatorik: Theorie der Vertretersysteme.
(April 1977)
- NR. 6 Mertens, P. :
Einige Hinweise zur technischen Gestaltung von
Bildschirmdialogen.
(Mai 1977)
- NR. 7 Jäpel, D. :
Ein Begriffssystem zur Formulierung von Schutzaspekten
und seine Anwendung auf Programmsysteme.
(Juni 1977)
- NR. 8 Brendel, W., Grötsch, E., Nagl, M., Schneider, H.-J. :
Graphersetzungssysteme, Beiträge zur Theorie, Anwen-
dung und Implementierung.
(Juni 1977)
- NR. 9 Rüdell, W. :
Darstellung und Vergleich abstrakter Modelle zur
Beschreibung geschützter Systeme.
(Juli 1977)
- NR. 10 Bolch, G., Gall, R., Grassl, M., Schmidt, B. :
Modelle für Trommelspeicher.
(Juli 1977)
- NR. 11 v. Eichhorn, K. :
Steiner Systeme, Codes und affine Räume.
(Juli 1977)
- NR. 12 Göttler, H. :
Zweistufige Graphmanipulationssysteme für die
Semantik von Programmiersprachen.
(Juli 1977)

Offsetdruck-Fotodruck

J. Hogl, 8520 Erlangen, Hauptstr. 109 (Nähe Martin-Luther-Platz) Tel. 22990